ESD-TR-72-101

Technical Note

1972-17

Design Study of the Advanced Signal Processor

P. E. Blankenship
B. Gold
P. G. McHugh
C. J. Weinstein

27 April 1972

Prepared for the Department of the Air Force
and the Advanced Research Projects Agency
under Electronic Systems Division Contract F19628-70-C-0230 by

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Lexington, Massachusetts

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA – R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Lincoln Laboratory, M.I.T.	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP None

3. REPORT TITLE

Design Study of the Advanced Signal Processor

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical Note

5. AUTHOR(S) (Last name, first name, initial)

Blankenship, Peter E.
Gold, Bernard
McHugh, Paul G.
Weinstein, Clifford J.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
27 April 1972	88	3

8a. CONTRACT OR GRANT NO. F19628-70-C-0230 b. PROJECT NO. 649L c. ARPA Order 1559 d.	9a. ORIGINATOR'S REPORT NUMBER(S) Technical Note 1972-17 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) ESD-TR-72-101

10. AVAILABILITY/LIMITATION NOTICES

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Air Force Systems Command, USAF Advanced Research Projects Agency, Department of Defense

13. ABSTRACT

A design study has been carried out for a general-purpose signal processing computer which incorporates arithmetic parallelism in a microprocessor structure. The study indicates that the processor (Advanced Signal Processor, ASP) would be faster, smaller, simpler, and less costly than its predecessor, the Fast Digital Processor (FDP). In addition, the ASP would have a more sophisticated in-out system than the FDP. These gains are achievable partially because of newly available fast hardware and partially due to the architecture of the ASP.

14. KEY WORDS

processor
advanced signal processor
signal processing computer

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LINCOLN LABORATORY

DESIGN STUDY OF THE ADVANCED SIGNAL PROCESSOR

P. E. BLANKENSHIP, B. GOLD, P. G. McHUGH, C. J. WEINSTEIN

Group 24

TECHNICAL NOTE 1972-17

27 APRIL 1972

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored in part by the Department of the Air Force under Contract F19628-70-C-0230 and in part by the Advanced Research Projects Agency of the Department of Defense under Contract F19628-70-C-0230 (ARPA Order 1559).

ABSTRACT

A design study has been carried out for a general-purpose signal processing computer which incorporates arithmetic parallelism in a microprocessor structure. The study indicates that the processor (Advanced Signal Processor, ASP) would be faster, smaller, simpler, and less costly than its predecessor, the Fast Digital Processor (FDP). In addition, the ASP would have a more sophisticated in-out system than the FDP. These gains are achievable partially because of newly available fast hardware and partially due to the architecture of the ASP.

Accepted for the Air Force
Joseph R. Waterman, Lt. Col., USAF
Chief, Lincoln Laboratory Project Office

CONTENTS

DESIGN STUDY OF THE ADVANCED SIGNAL PROCESSOR

I. INTRODUCTION

Applications such as radar, speech analysis and synthesis, and sonar, require a great number of signal processing operations to implement a system. The advantages of carrying out these operations digitally in real time have become well established in recent years. This design study describes the Advanced Signal Processor (ASP), a fast programmable signal processor that can be integrated into a real-time system for these and other applications. It is emphasized that this report is a design study and does not describe a machine that has been built. Plans for construction of the ASP are indefinite at this time.

Speed is the prerequisite in a real-time system. The key features of the ASP are speed, programmability, communications, and compactness. The ASP will be slightly faster than the Fast Digital Processor (FDP),[1] a Laboratory computer that has speed enough for real-time radar and speech applications. Like the FDP, the ASP is a general-purpose processor so that rapid spectral analysis and other signal processing functions such as windowing, magnitude taking, and thresholding can be implemented by programming.

The ASP will differ from the FDP in communications capability and size. The FDP was designed as part of a Laboratory computing facility; the ASP has been designed to serve as part of a real (though perhaps experimental) system. The FDP was given only minimum (input-output) communications capability. Complicated communications is handled by the nearby Univac 1219 computer. Experience gained in integrating the FDP into a real radar system has indicated that a more sophisticated in-out system would have been quite desirable. Such a system will be incorporated into the ASP to facilitate communications with external memories, other computers, and various other devices. Also, the FDP is large and immobile, but the ASP will fit into a medium sized airplane while retaining and actually surpassing the FDP's processing power.

As orientation to a description of the ASP, this design study begins by noting the relationship of the ASP to the FDP and the LX-1 microprocessor,[2] two general-purpose processors built at the Laboratory. Then the main instruction classes and their execution times are described. (Detailed descriptions of the instructions appear in the Appendix.) The programming features of the ASP, which make it attractive for signal processing applications, are illustrated by examples. Important hardware features of the ASP are also described.

II. ASP STRUCTURE

The ASP's structure was motivated largely by a consideration of the assets and liabilities of the FDP and the LX-1.

A. Features of the FDP and LX-1

The FDP was designed as a general-purpose processor which could perform signal processing operations such as spectral analysis and digital filtering about 100 times faster than with standard computers. The FDP, whose structure is shown in Fig. 1, derives its speed from three basic factors - arithmetic parallelism, instruction cycle overlap, and

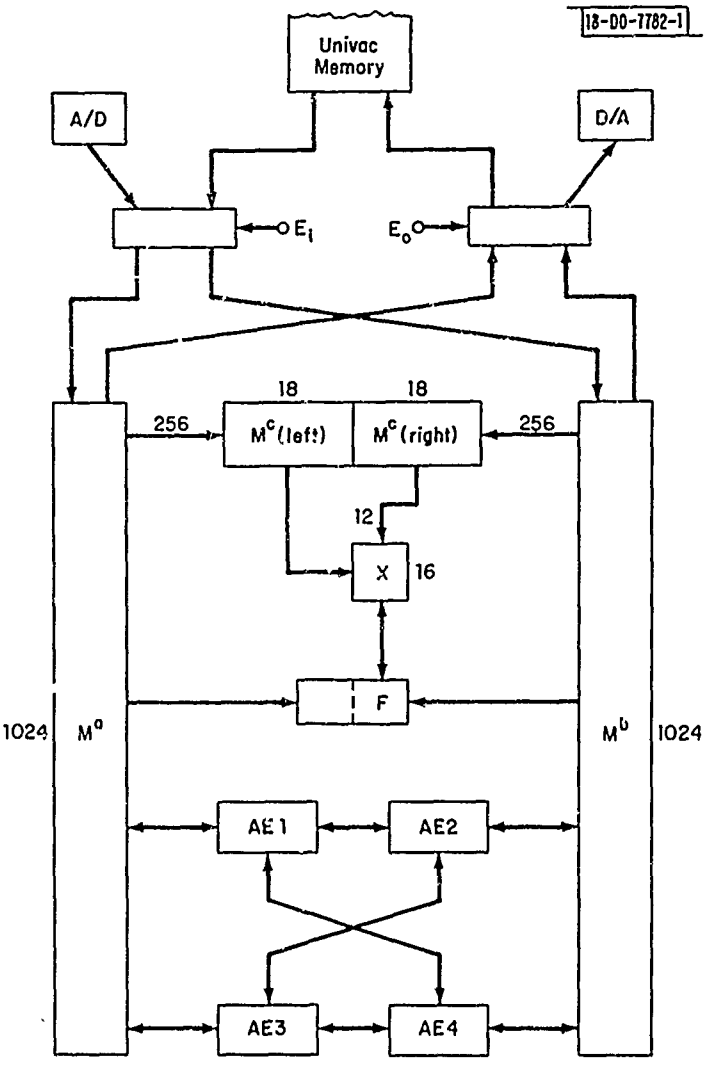


Fig. 1. Structure of the Fast Digital Processor.

fast hardware. The four arithmetic elements (AEs) can operate in parallel under independent control. Each contains an 18 x 18 array multiplier in addition to adder and logic function hardware. The program memory M^c is separate from the data memories M^a and M^b, and a three-level overlap of instructions is carried out. While a typical instruction is being executed, the next instruction is being decoded, and a third instruction is being fetched. The FDP was built from Motorola MECL II integrated circuits, the fastest logic line available at the time of design. The speed of the FDP is such that Doppler processing for 2048 range gates, including a 64-point fast Fourier transform (FFT) and various other operations for each range gate, could be performed in about two seconds. These operations are being carried out by the FDP in a real-time demonstration radar system.

This speed is the key asset of the FDP and ought to be retained, and if possible, augmented in a new processor. However, an important liability of the FDP is its great complexity and associated large size and cost. The physical construction of the FDP was designed for engineering accessibility rather than small size, but even with repackaging the FDP would remain too large for, say, an airborne radar application. A desirable goal is to retain or augment the FDP's speed in a significantly smaller and less complex machine. Three important aspects of the FDP, which contribute to its large size, have been modified in the ASP. First, the basic word length of 18 bits was found to be more than necessary for the demonstration radar and similar applications. The ASP will use a basic 12-bit word length but will allow fast 24-bit operations when desired. Second, the number of arithmetic units in the ASP is cut down to allow only two-fold arithmetic parallelism. Third, the FDP has very complex control (for example, all AEs are controlled independently) and a large number of specialized data paths such as those between AE's and those between the various special registers internal to each AE.

The in-out capability of the FDP was made quite limited since it was expected that the UNIVAC 1219 would handle much of the required I-O. Experience with implementation of an actual range-gated Doppler radar has

indicated that a slightly more sophisticated I-O system would be desirable, and such a system will be included in the ASP.

The LX-1 microprocessor is a general-purpose computer whose present chief application is to display processing; it was not designed especially for signal processing. However, the LX-1 is an inherently simple and small machine yet has some features which are quite attractive for a signal processing computer. In Fig. 2 the LX-1 is shown to contain a set of general

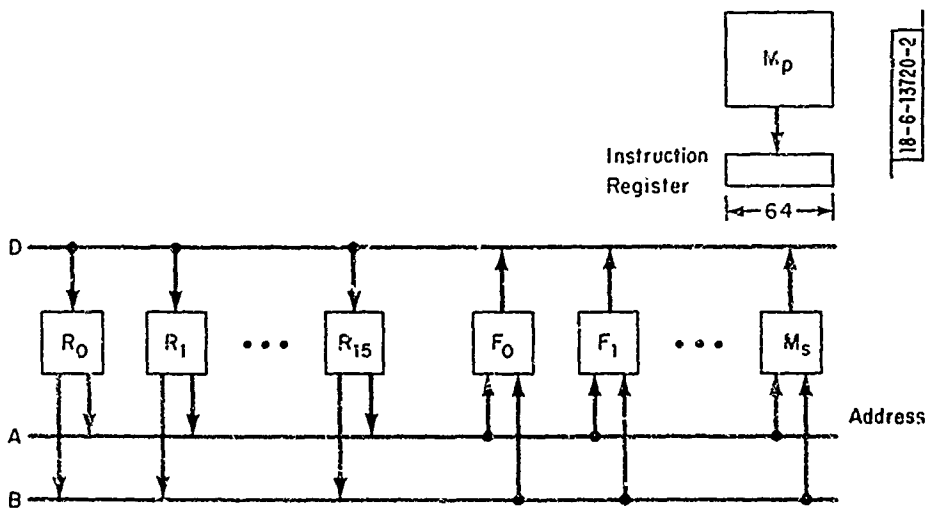


Fig. 2. LX-1 microprocessor.

registers R_i, a set of function boxes F_i, a data memory M_s, and three busses A, B, and D which interconnect these parts. The basic data word is 16 bits long. The control resides in the program memory M_p. In a typical function instruction, two registers, say R_3 and R_7, are read onto the A and B busses, an operation such as multiplication is performed in one of the function boxes, and the result is written from the D-bus into another general register, say R_5. For a memory instruction, M_s is addressed from the contents of a register placed on the B-bus, and reads from the A-bus or writes onto the D-bus. Machine control is quite simple, since all instructions cause data to flow through the busses in a similar way and there are no specialized paths between certain special registers. Also programming of the LX-1 is quite simple because it is a serial machine.

A key feature of the LX-1, which differentiates it from the FDP as well as from standard computers, is the set of general registers. These general registers have great flexibility, being useful, for example, as index registers or arithmetic accumulators. The fact that all general registers are accessible in the same way to the busses and the function boxes serves to minimize the data shuffling necessary during a computation. For example, in an FFT butterfly programmed on the FDP a number of instructions must be devoted to shuffling data between the various specialized I, Q, and R registers.

The LX-1 however, is significantly slower than the FDP in signal processing applications. The LX-1 has fast hardware, but since it lacks parallelism is only about one-fourth as fast as the FDP for an FFT butterfly.

B. ASP Architecture

The ASP architecture represents a synthesis of some of the speed-producing parallelism of the FDP into an LX-1 type structure featuring general registers, simplicity of architecture and control leading to a small size potential, and simplicity of programming. A new line of hardware, faster than was available for the earlier machines, will be used.

The structure of the ASP, depicted in Fig. 3, features like the LX-1 a set of general registers M_r, function boxes, a data memory M_s, a bussing structure, and a program memory M_p. However, several key departures from the LX-1 are to be noted. The busses carry 24-bit words that may be separated into two 12-bit bytes. The function boxes have dual sets of 12-bit arithmetic hardware so that, for example, in the adder function box, two simultaneous 12-bit adds or one 24-bit add can be carried out as a single instruction. With the configuration box, which allows swapping of the two 12-bit bytes of a word, and an inhibition option, which allows nullification of either of the two dual operations, completely flexible manipulation of the bytes is possible.

The ASP will have 64 24-bit general registers (compared to 16 for the LX-1). This large number of general registers provides a very high-speed temporary storage, which as will be illustrated later, can be used to speed

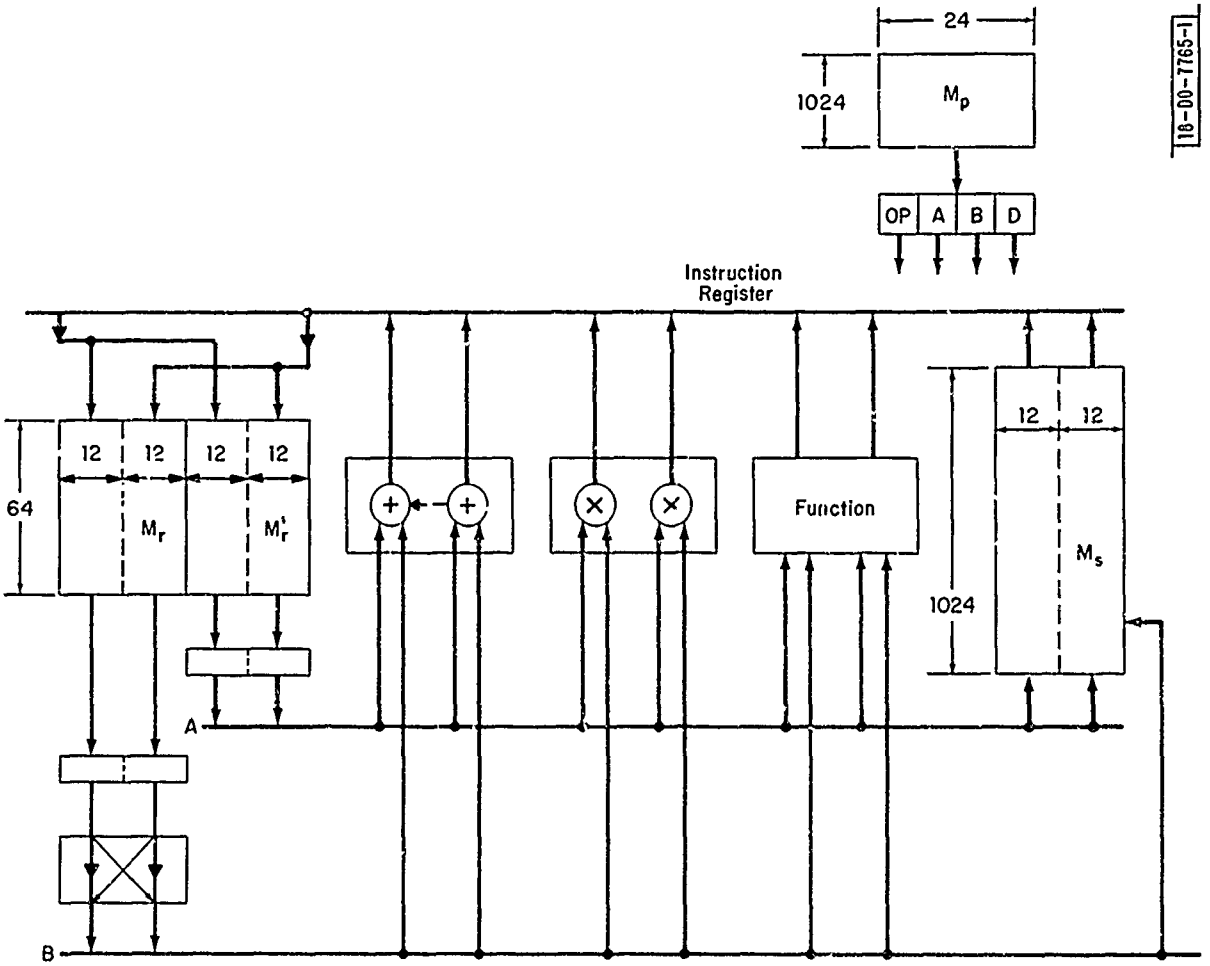


Fig. 3. Structure of the Advanced Signal Processor.

up signal processing programs. The large number of general registers are feasible because of the availability of very fast integrated circuit memories which permit the general registers to be realized as a memory, rather than as a set of separate flip-flop registers, with negligible loss in speed. Since two operands must be read from M_p in each instruction, the two physical memories, M_r and M'_r, will contain identical contents and be read simultaneously.

The program and data memories are each 1024 x 24 integrated circuit memories. Some overlap between reading of M_p and execution of instructions will be incorporated.

The function boxes of the ASP will include: (1) an adder-logic complex capable of performing two 12-bit or one 24-bit add, subtract, or logic operation per instruction; (2) a multiplier box containing two 12 x 12 array multipliers; (3) a special function box to facilitate shift and normalization operations; and (4) an array divider. Like the LX-1, the ASP has a highly modular structure so that different versions of the machine could include new function boxes or leave out some of those just listed.

The in-out system of the ASP will include a pair of 24-bit direct memory access channels each of which can provide data flow to and from M_s in parallel with the main program. There will be six additional auxiliary channels to allow control signals (but not data) to be transmitted to, and received from, other devices.

III. INSTRUCTION REPERTOIRE

The main instruction classes available in the ASP are now introduced. Instruction word formats will be presented, and some examples of particular instructions and their execution times will be given. A detailed listing with definitions of the instruction set, as it currently stands, is provided in the appendix.

A. Arithmetic and Logical Operations

In this class are included all the instructions which are executed in the adder-logic function box. All arithmetic in the ASP is 2's complement. Three different instruction formats are utilized to control the adder-logic functions, as indicated in Fig. 4.

In the 3-field instructions, A selects one of 64 operands from M_r for the A bus, B selects one of 64 operands for the B bus, and D

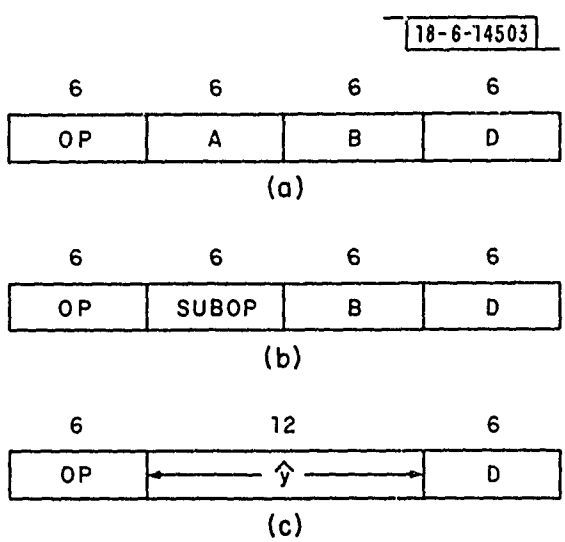


Fig. 4. Three instruction formats to control adder-logic functions: (a) 3-field format, (b) 2-field format, (c) 1-field format.

selects one of 64 destinations in M_r for the result. All operands are 24 bits, but the options of configuration and inhibition allow flexible operation on 12-bit bytes. For example a typical instruction can perform the operations:

$$A_u + B_\ell \rightarrow D_u \; ; \; A_\ell + B_u \rightarrow D_\ell$$

where the subscripts u and ℓ refer to upper and lower 12-bit bytes, respectively. Such an instruction, consisting of two 12-bit adds, can be performed in 65 nsec. In addition to various manipulation of the bytes, some scaling provision is included.

The pair of operations

$$(1/2)(A_u + B_u) \rightarrow D_u \; ; \; (1/2)(A_\ell + B_\ell) \rightarrow D_\ell$$

can be executed in 65 nsec. The option for 24-bit arithmetic is included, and the 24-bit add

$$A + B \rightarrow D$$

can be executed in 75 nsec. Also included among the 3-field format instructions are the bit-by-bit logic operations AND, XOR, and IOR, each of which takes 65 nsec. The FDP cycle time is 150 nsec for all instructions except the multiply, which takes 450 nsec.

In the 2-field format, which is included to allow more option codes than would be otherwise possible, the A-field is ostensibly missing, but the A operand is taken as the same general register as the D destination. This group consists of various other adder-subtractor options which are differentiated according to configuration, inhibition, scaling, and single or double precision. An instruction type of interest is the sign extended add which permits, for example, B_ℓ to be sign extended to 24 bits and added to the 24-bit A operand. Another noteworthy instruction is the zero inject instruction, which shifts B_ℓ right one place and unconditionally forces a zero into the sign bit. This instruction is quite useful in programming a 24 x 24 bit multiply. Finally, a bit reversed add instruction similar to that in the FDP, is included.

The 1-field format is used for operations with 12-bit constants $\hat{y}$ which comprise part of the instruction word. The constant can be inserted

in or added to either half of the M_r register addressed by the D-field.

B. Multiplication

The two array multipliers in the multiplier function box each perform signed 12 x 12 bit multiplies yielding 24 bits of product. All multiply instructions are executed in 120 nsec. Various options are provided as to the configuration of the input bytes and the possible inhibition of writing either of the two multiplier outputs. Also options are provided to select those bits of the 24-bit products that are to be transmitted to the two 12-bit output bytes.

C. Division

The divide box contains an array divider and is capable of dividing a 24-bit dividend by a 12-bit division and proving a 12-bit quotient in 220 nsec.

D. Scaling

The scaling functions are designed to be used in conjunction with the multiplier to yield efficient programming of normalization and shifting. For example, to left justify a number, one would use the scale function (SF) instruction to determine the necessary number of places to shift, the scale factor positive (SFACP) instruction to set up a multiplier to effect the shift, and a multiply instruction to actually carry out the shift. The entire normalization would take 65 + 65 + 120 = 250 nsec and could be used in floating point operations as well as in block normalization. The scaling operations included are quite simple and require much less hardware than that needed in a complete shifting matrix. The fast multiply permits shifting to be accomplished quite ·ickly without such a matrix.

E. Memory

The memory reference instructions have the 2-field format of Fig. 4b. The B-field points to the M_r location, which contains the M_s address of interest, and the D-field points to the source for writing or the data destination for reading. The various memory instruction options permit the M_s address to come from B_u or B_ℓ and the data source or destination to be either D_u or D_ℓ for a 12-bit transfer, or D for a 24-bit transfer. The time

for a memory read instruction, which includes the required accesses to M_r and M_s, is 100 nsec. The time for a memory write instruction is 80 nsec.

F. Branching

The branching instructions in the ASP include arithmetic jumps, overflow jumps, unconditional jumps, a jump conditional on in-out activity, and a skip make instruction.

Arithmetic jumps are conditional on the contents of selected M_r registers. For example, one may jump on the condition that the upper byte of some M_r register is positive, or on the condition that the full 24-bit word in a specified M_r location is zero. The arithmetic jumps may be used with or without a skip on jump (SOJ) like that in the FDP. If the SOJ bit in the jump instruction word is not set, the instruction after the jump will be executed even if the jump condition is met. If the SOJ bit is set, the instruction after the jump will be nullified if the jump condition is met. The "SOJ not set" option would save time in a tight loop since one instruction cycle time is effectively lost in killing the next instruction. The format for arithmetic jumps is indicated in Fig. 5. D selects the M_r register to be tested and B selects upper or lower byte; $\hat{y}$ selects the M_p location to be jumped to; and α specifies the SOJ option.

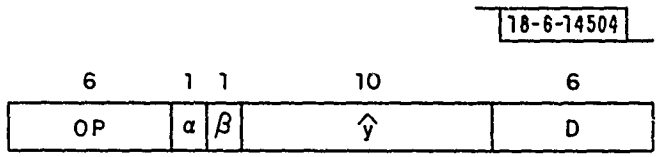


Fig. 5. Format for arithmetic jumps.

The overflow jumps are similar to the arithmetic jumps except that the format is different (since a D-field is not needed) and the jump conditions are the various types of overflow that can result from arithmetic operations. The in-out activity jump tests various activity conditions on an in-out channel. The skip make instruction is patterned after that in the FDP and allows skipping of any combination of the next four instructions according to the conditions in one of the 16 flags in the ASP.

G. Input-Output and Block Transfer

The I-O system of the ASP includes two 24-bit direct memory access data channels and six control channels, each of which can be monitored on an interrupt basis.

Each data channel provides 24 input data lines, 24 output lines, and several control lines to carry request signals and mode information. Data transfers are initiated by a DMA instruction, which transmits to the I-O hardware such parameters as block size and starting M_s address. The I-O channel hardware then carries out all operations needed to effect the transfer, slowing down the main program only when both require access to M_s at the same time. When the buffer is complete a monitor interrupt (if desired) then causes the main program to jump to a service routine whose location was also specified in the DMA instruction.

The six control channels are identical to the data channels except that the data lines are omitted. The control signals could synchronize the ASP with other computers in a real-time system.

The block transfer instruction (BLK) transfers a list of words from M_s into M_p and is quite similar to the corresponding instruction in the FDP. Like DMA, BLK must specify a block size and starting addresses. But unlike DMA, the BLK causes all other operations to cease during its execution.

IV. PROGRAMMING FEATURES

Some examples of the ASP's programming features:

A. Double Precision and Floating Point

The ASP was designed so that 24-bit, fixed point arithmetic could be performed quite efficiently. A 24-bit add or subtract is performed in a single 75-nsec instruction, and a 24-bit memory access is accomplished with one memory instruction. Of course the machine's dual parallelism is lost for 24-bit operations. A 24 x 24 bit multiply must be programmed. Using the formula

$$AB \approx A_u B_u + 2^{-11} (A_u B_\ell' + B_u A_\ell'),$$

where A_ℓ' or B_ℓ' is formed by shifting the lower byte of A or B right one bit and forcing the sign bit to zero (the ZINJ instruction), a result accurate to 22 bits can be obtained in six instructions or 520 nsec.

There are no hardware floating point instructions on the ASP, and floating point arithmetic must be programmed. However, the scaling functions mentioned above facilitate the shifting and normalizations needed for floating point. A single precision (12-bit fraction, 12-bit exponent) floating point multiply can be executed in about 0.7 μsec, while a double precision (24-bit fraction, 12-bit exponent) multiply takes about 1.1 μsec. Single precision floating add takes about 1.6 μsec while double precision requires 2.7 μsec. These times seem slow in comparison to fixed point operations, but compare favorably with other computers. For example, the IBM 360 Model 67, which has hardware floating point takes about 5 μsec for a multiply and 2.5 μsec for an add. Standard computers without floating point hardware take significantly longer.

B. FFT Butterfly

The basic computation in an FFT is the so-called butterfly computation which, as indicated in Fig. 6, operates on two complex numbers to

18-DO-7976-1

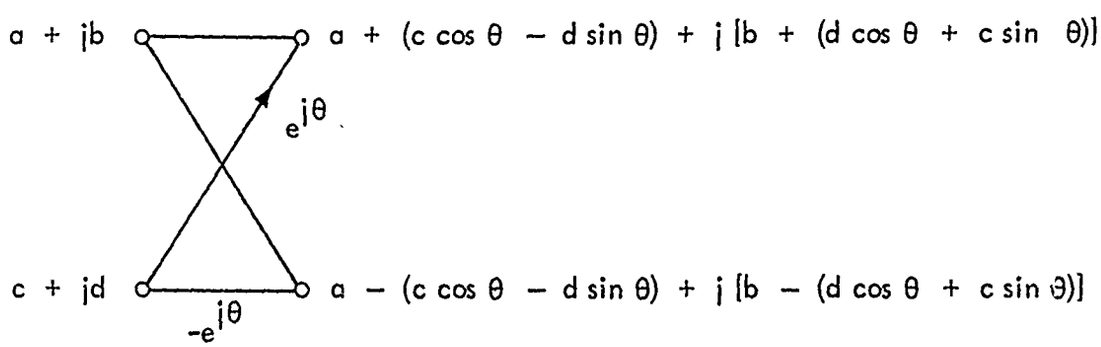


Fig. 6. Butterfly computation: FDP--10 instructions, 1.5 μsec; LX-1--30 instructions, 4.5 μsec; ASP--12 instructions, 1.0 μsec.

yield two new complex numbers and requires a complex multiply and two complex adds. A standard N-point radix 2 FFT requires $(N/2)\log_2 N$ butterfly computations. A butterfly can be programmed on the ASP with 12 instructions, including 4 memory accesses, 3 add instructions, 12 multiply instructions, and 3 index and branch instructions. The execution time is about 1.0 μsec. For comparison, a butterfly on the FDP, as programmed for the demonstration radar, takes 10 instructions or 1.5 μsec.

Thus the ASP will be somewhat faster than the FDP for standard FFT programs. This speed-up comes chiefly from the faster instruction execution resulting from the faster hardware, and the fact that 12-bit operations take less time than 18-bit operations.

C. Radix 8 FFT

The foregoing discussion indicated the speed of the ASP in carrying out a standard radix 2 FFT. By means of slightly more sophisticated FFT programming, advantage can be taken of the large number of fast general registers to achieve significant speed-up.

The technique can be illustrated by the example of a 64-point FFT, programmed in radix 8. The input data in M_s is thought of as organized in a two-dimensional array as depicted in Fig. 7. The FFT is begun by bringing the first row into M_r and computing an 8-point FFT of this row without additional access to M_s. The 8-point FFT is implemented as efficiently as possible; for example, when the coefficient $e^{j\theta}$ in the butterfly is 1 or j (more than half the cases), no multiplications are executed. Each of the eight outputs is multiplied by a complex twiddle factor, and the

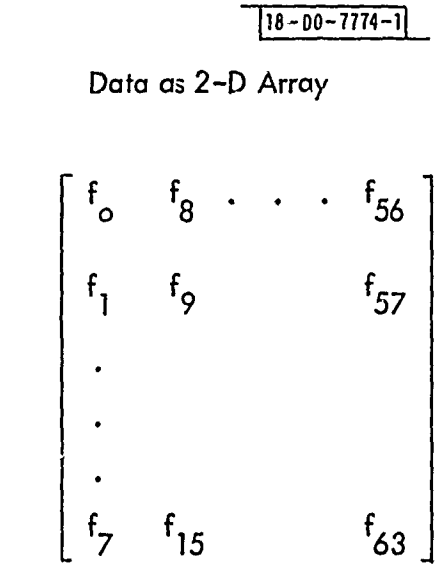


Fig. 7. 64-point FFT, radix 8. Computational steps: (1) eight 8-point discrete Fourier transforms (DFT) on rows, (2) twiddle factors (64 complex multiplies), (3) eight 8-point DFTs on columns.

results are stored back in place in M_s. This procedure is repeated for all the rows. Then each column is brought into M_r, transformed (no additional twiddle factors are necessary), and stored back in M_s. This completes the 64-point FFT.

In this implementation of a 64-point FFT, only two exchanges of the array between data memory and the general registers are necessary. This saves two-thirds of the memory access time of a radix 2 algorithm, which requires $\log_2 64 = 6$ such exchanges. Also the 8-point FFTs may be coded more efficiently by eliminating unnecessary multiplications. The result is that, with radix 8, a 64-point FFT can be computed in about 60% of the time necessary for a radix 2 program. This saving is possible only because there are enough general registers to provide all the necessary storage for an 8-point FFT.

This technique can be extended to FFTs of other sizes, and implementation with other radixes. Also the general technique of using M_r as high speed temporary storage can speed up a wide variety of programs.

D. Large FFT with External Core Storage

The high-speed data memory of the ASP will be initially limited to 1024 words because of size and cost considerations. However, it is often desired to perform an FFT where the number of samples is too large to be accommodated in M_s, and it would be advantageous if such a transform could be carried out with only small speed loss caused by shuffling the data in and out of an external core memory. The direct memory access capability of the ASP makes this possible. The technique will now be illustrated by a 2048-point FFT example.

Consider the data (stored sequentially in core) as a two-dimensional, 32 x 64 array where the rows consist of samples spaced by 32 sampling intervals and the columns contain sequential samples. A 64-point FFT on each row is computed, and the results are multiplied element-by-element by a set of complex constants (called twiddle factors, and which are also stored in core) and stored back in core. Then 32-point FFTs on each column are performed and the computation is complete. The transform will be ordered

in core with rows and columns interchanged.

While the processor is computing the FFT of a row, the next row of data and twiddle factors is flowing into data memory and the last computed row is being sent to core by means of direct memory access block transfers which are controlled by in-out hardware and slow down the FFT computation only negligibly. Associated with the core memory must be an address box, which, after initiation from the processor, can sequence through an arbitrary number of core locations with an arbitrary spacing between locations.

With the scheme just sketched out, the 2048-point transform can be computed essentially as fast as if sufficient fast memory were available to store the entire array.

V. HARDWARE FEATURES

This section describes the processor's logical design and its method of construction as now envisioned. Changes can be expected as the design progresses.

The treatment begins by explaining why MECL 10K integrated circuit logic units were selected for building the processor. The general registers, function boxes, timing, input-output, remote console, and construction are then described. The processor's control circuitry is not yet defined.

A detailed description of the processor's instructions is given in the Appendix and a familiarity with them is assumed.

A. MECL 10K

The basic ground rules for choosing an integrated circuit logic line for the processor were that using it we could produce a machine with an instruction cycle time less than 100 nsec that could be packaged in a 6-ft relay rack. The machine's speed and physical size were estimated by studying designs of multipliers, general registers, and memories. Three logic lines were considered: Schottky T^2L, 1-nsec Emitter Coupled Logic (ECL), and 2-nsec ECL. Schottky T^2L was eliminated because of speed; its gate delay is 3 nsec which is 50 percent slower than the 2-nsec ECL. The 1-nsec ECL line, Motorola MECL III has a limited number of logic functions, must be

packaged on multilayer boards, and its gates dissipate almost twice the power of 2-nsec ECL circuits. The 2-nsec ECL was selected.

Two 2-nsec ECL lines are commercially available at this time: Motorola MECL 10K and Fairchild 9500. Both lines are equal in speed and contain, or will contain, equivalent circuit functions. Tentatively, the new processor will use Motorola MECL 10K because: (1) Requires less power, 30- vs 75-mw/gate when driving a 2-KΩ load. (2) Compatible output voltage levels with the voltage levels of Advanced Memory Systems (AMS) memory element over the temperature range 0° to 70°C. The processor's memories and general registers are to be built from the AMS circuit. (3) Four-bit arithmetic logic element, vital for array multipliers and dividers, is currently available in quantity.

Some of the MECL 10K line's more important features are:

(1) 2-nsec propagation delay and 3-nsec rise and fall times
The 3-nsec rise and fall times are slow enough to permit unterminated lines up to 4 in. long without worrying about reflections. The 1.2-nsec rise and fall times of MECL III restrict line lengths to less than 1 1/4 in.

(2) 50 Ω drive capability
For lines longer than 4 in. where reflections are a problem, the lines can be terminated in 50 Ω or greater to negate reflections.

(3) Balanced twisted pair line interface
Signals can be transmitted and received over balanced twisted pair, a good way to distribute the system clock because it is easy to control the transmission delays by changing line lengths.

(4) Compatibility with MECL II and III
Compatibility with high speed MECL III and slow speed, 4-nsec propagation delay MECL II provides the MECL 10K line added flexibility.

A semiconductor memory is necessary to realize the proposed machine. Unfortunately, neither Motorola nor Fairchild have an off-the-shelf memory element whose speed is compatible with the rest of the circuits in

their respective lines. Fortunately, Advanced Memory Systems (AMS) is producing a 64-bit memory element that is compatible with MECL 10K, though not compatible with Fairchild 9500. The element is organized as 64 words, 1 bit per word, and has a 7-nsec read time and a 7-nsec write time.

MECL 10K was selected instead of Fairchild 9500 because of the availability of MECL 4-bit arithmetic logic circuits and compatibility with the AMS memory circuits.

B. General Registers

Reviewing how the processor's structure works (Fig. 8), assume that a 12-bit add instruction has just been transferred into the instruction register from the program memory. The 6-bit A address selects a 24-bit general register whose output is transferred onto the A bus, and the 6-bit B address selects a second 24-bit general register whose output is transferred onto the B bus. The 12-bit add is executed in the adder function box, using the A and B operands which are available at its inputs. The result is stored in the general register specified by the 6-bit D address.

The whole operation has three distinct parts: (1) read the general registers, (2) execute instruction in a function box, and (3) write result back into general registers.

It is apparent from this review that an instruction's speed is highly dependent on how fast the general registers can be read and written. Even if an add or multiply could be executed in zero time, a complete add or multiply instruction would still require time to read and write the general registers.

The general registers can be realized in two ways. In both designs, the A and B operands will be read from the general registers simultaneously instead of serially to increase the speed at which instructions can be executed.

The logic needed to build the general registers from flip-flops (FF) and gates is indicated in Fig. 8b. This design has two major problems besides requiring 64 separate FF registers: (1) One bit of the bus is obtained by multiplexing together 64 FF outputs, and this must be done for

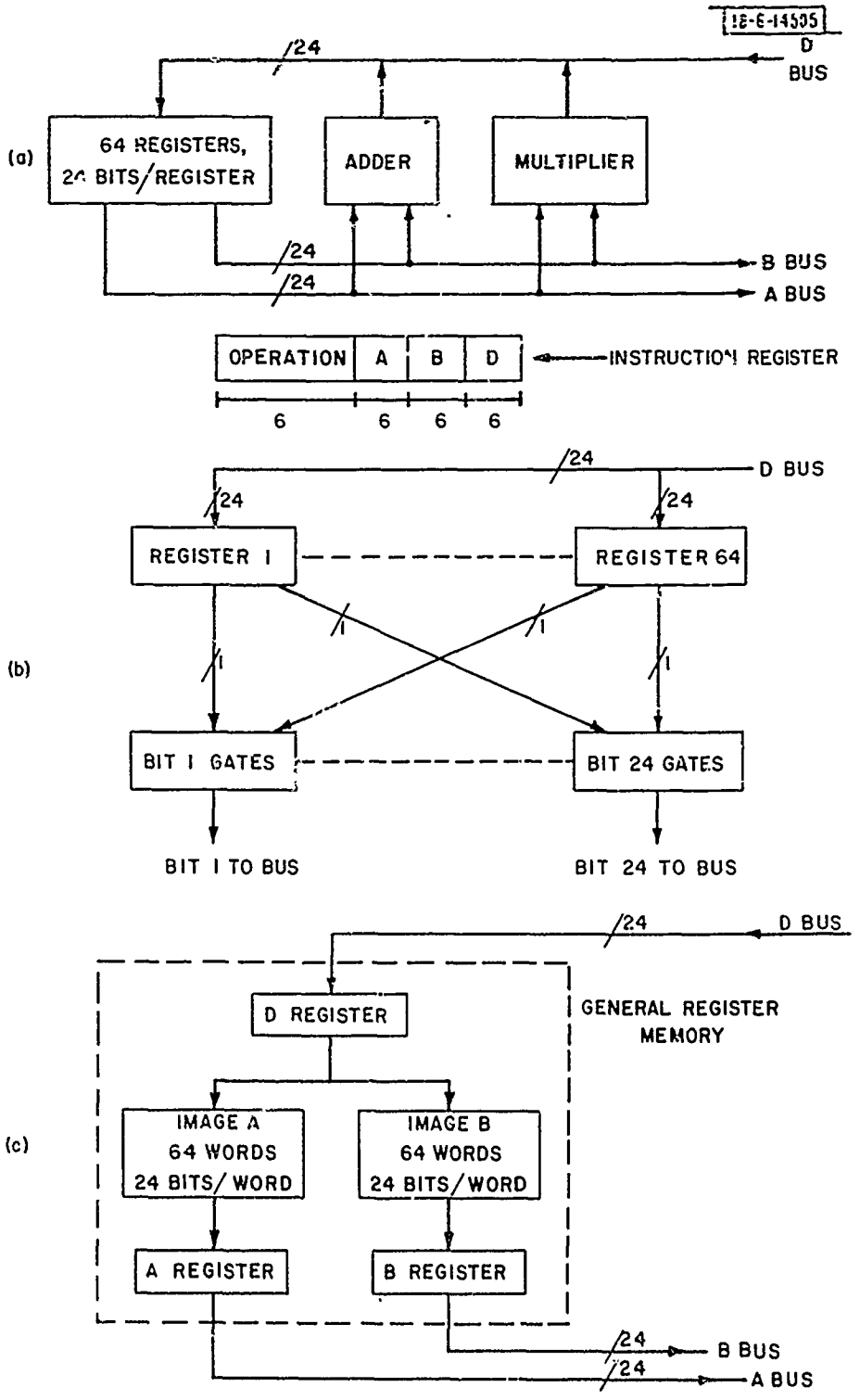


Fig. 8. General registers.

each of the 24 A and 24 B bus bits; (2) Each of the 24 D bus lines must be distributed to 64 loads.

This design would use over 2000 MECL 10K packages — an excessive number for a small machine.

The general registers (Fig. 8c) for this processor contain two 64-word, 24 bits/word image memories, which word-for-word always contain identical data. The A operand is read from the A image, and simultaneously, the B operand is read from the B image. Both operands are stored in bus registers before sent to the function boxes.

Results are written into the two memories by storing them temporarily in the D register. When the memories are not busy, e.g., when an add or multiply is actually being performed in a function box, the contents of the D register can be written into both memories. The write operation does not affect the other function boxes because they are isolated from the memories by the bus registers. This method of writing the memories permits "burying" or hiding the time needed to write them, a minimum of 7-nsec of memory element write time.

It takes 150 integrated circuits that include 48 64-bit memory elements to build these general registers, which is considerably less costly than the previous solution (Fig. 8b).

When the Next Instruction Pulse (NI Pulse) is generated by the processor's control circuitry, which is not shown in Fig. 9, a new instruction is transferred into the instruction register. Simultaneously, contents of the D bus, which is the result of the last instruction and may or may not have meaning, are transferred into both the DA and DB registers. Also, the 6-bit D address portion of the instruction register, which specifies the address at which the contents of the D bus will be stored in the image memories, is transferred to the Store Address Register.

Data are now read from the A and B image memories by addressing them with the new A and B addresses, which are in the instruction register. In parallel, the two addresses are compared with the store address. If either address is equal to the Store Address, and if the last instruction

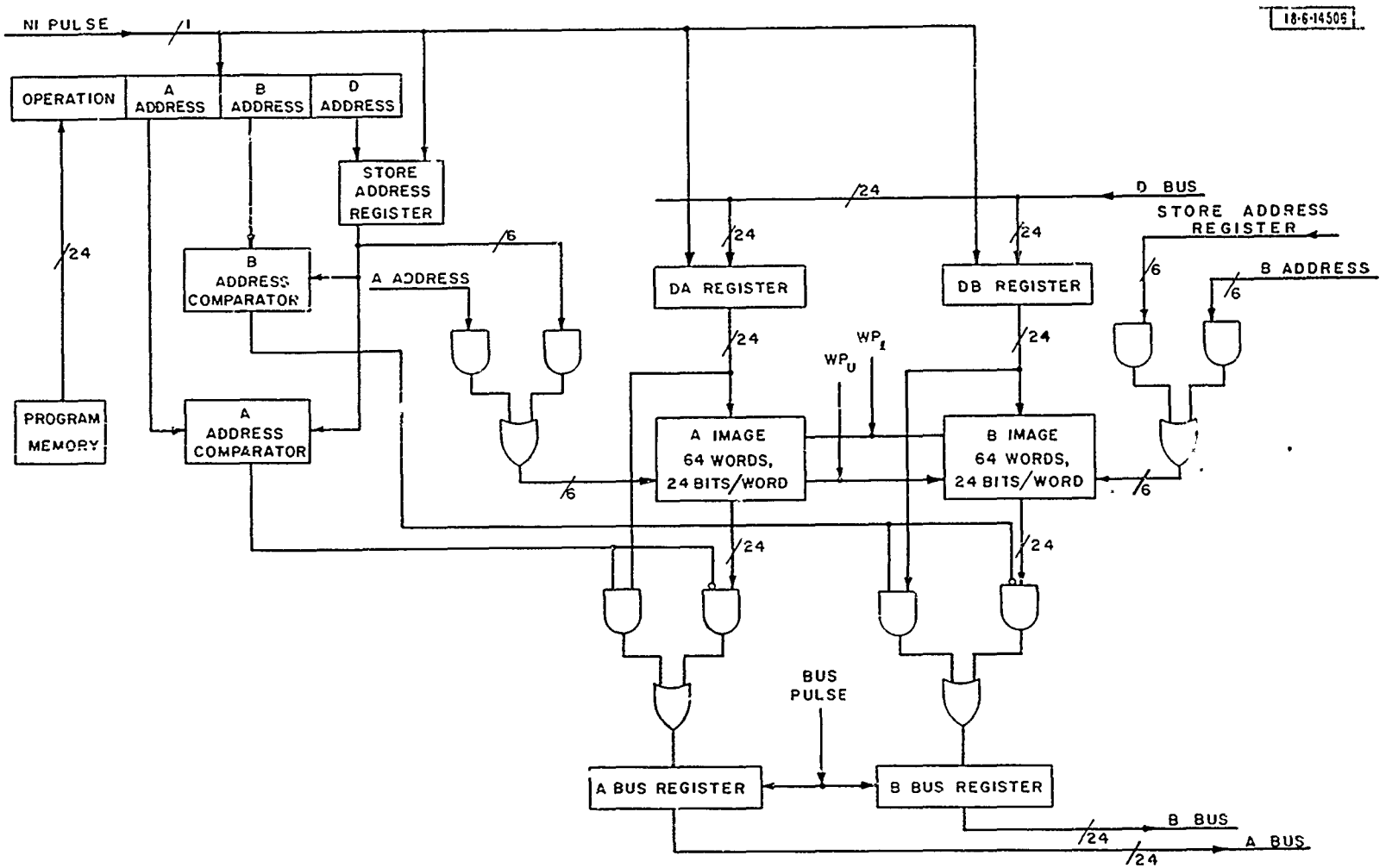


Fig. 9. General register memory.

produced a result that must be stored in the general register memory, a one level is produced at the appropriate comparator output indicating that the needed operand is stored in the DA and DB registers and has, as yet, not been written into the image memories. If neither comparator output is a one, the outputs of the image memories are switched through the bus input gating circuitry and transferred into the bus registers when the control logic generates a bus pulse. When an operand address is equal to the store address, the appropriate address comparator output will be a one and this will switch the correct data storage register, DA or DB, through the bus input gating logic and into the bus register when the bus pulse occurs.

The outputs of the bus registers go to all function boxes and they are transformed in the particular function box, which is specified by the operation code of the current instruction. In parallel, data in the DA and DB registers are written into the image memories, which are not now involved in the function box operations at the location specified by the Store Address Register. wp_ℓ and wp_u are the memory write commands; wp_ℓ initiates a write operation in the lower 12 bits of a storage location, and wp_u initiates a write operation in the upper 12 bits. If the last instruction produced a 24-bit result, both wp_ℓ and wp_u will be enabled. If the last instruction produced a 12-bit result, either wp_ℓ or wp_u will be enabled; the choice between wp_ℓ or wp_u depends on whether the 12-bit result appears in the lower or the upper half of the 24-bit word. If the result is in the lower 12 bits of the word, wp_ℓ is enabled; if it is in the upper 12 bits, wp_u is enabled.

Read time is defined as that interval which begins when new data are transferred into the instruction register and which ends when the A and B operands arrive at the function box inputs. Thus read time for the image memory realization is 30 nsec.

The component propagation delays for the logic in Fig. 9, that result in a 30-nsec read time, as defined, are shown in Fig. 10, a simplified general register timing diagram. The actual time to read the image memories, 7 nsec, is less than 25 percent of the 30-nsec general register read time.

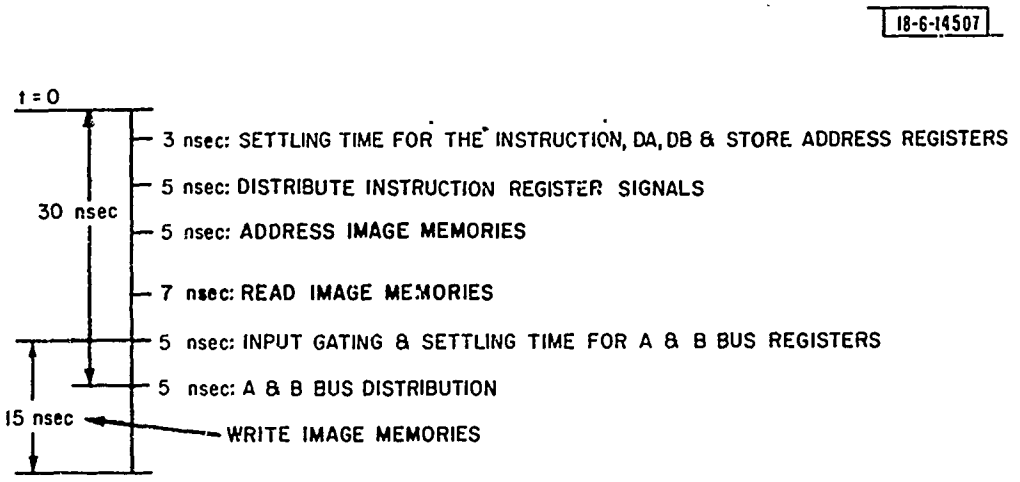


Fig. 10. General register memory timing.

C. ALU Function Box

The arithmetic logic unit (ALU) function box contains logic to implement the following types of instructions:

(1) Single 12-bit, and double 24-bit precision additions and subtractions

(2) Double-precision logical operations

(3) Modification of general register by constants

(4) Special functions: bit-reversed add, scale function, scale factor positive, scale factor negative, zero inject

(5) Branching.

These instructions are explained in detail in the Appendix and in Section H.

The logic needed to implement these instructions except for the special scale functions can be realized with a versatile adder-subtractor-logic unit, which has two nearly identical halves called 12-bit adders (Fig. 11). The basic adder element is the MC 10181, the 4-bit ALU. Two 4-bit numbers and a carry are entered into each ALU element and four sum bits and a carryout as well as some important auxiliary functions are produced. When the A and B inputs to an ALU element change, it takes 7 nsec for the element's sum outputs to stabilize and 5.4 nsec for its carry output to stabilize. When an element's carry input changes while its A and B inputs are static,

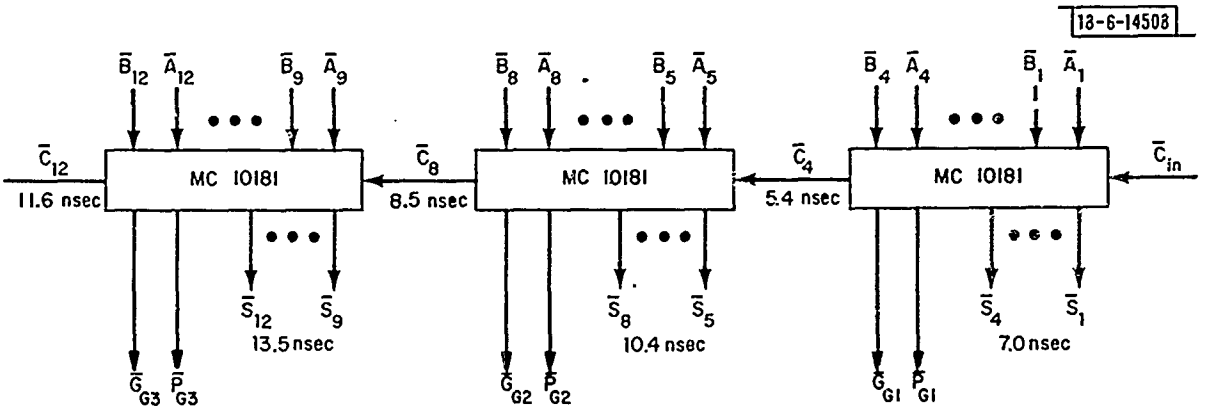


Fig. 11. 12-bit adder.

it takes 5 nsec for the element's sum outputs to settle down and 3.1 nsec for its carry output to settle down. Using the ALU, it takes 13.5 nsec to add two 12-bit numbers.

Although not shown in Fig. 11, each ALU has five control inputs, which choose which one of 32 possible functions (16 logical, 16 arithmetic) the ALU will perform. Examples of the logical functions are the logical product, A · B, the logical "Or", A+B, A itself, or B itself. Examples of the arithmetic functions, besides addition and subtraction, are the function 2A and A plus A · B. All of these functions are performed in a time equal to or less than an add.

Double-precision operations are performed in a 24-bit adder, built by interconnecting two 12-bit adders. The input to the second 12-bit adder is $\overline{C}_{12}$, the carry out of the first 12-bit adder. $\overline{C}_{12}$ is generated in 7.3 nsec in the fast carry circuit (Fig. 12b) instead of taking it directly from the carry out of the first 12-bit adder. The fast carry circuit uses the ALU element's $\overline{P}_G$ and $\overline{G}_G$ functions, which are defined in Fig. 12a. It takes 5 and 3 nsec for $\overline{G}_G$ and $\overline{P}_G$, respectively, to stabilize after an input variable change. The complete 24-bit add requires 20.7 nsec: 7.3 nsec to generate C_{12}, 2.2 nsec to gate C_{12} into the carry input of the first stage of the second adder, and 11.2 nsec for the second adder to produce its 12-bit result.

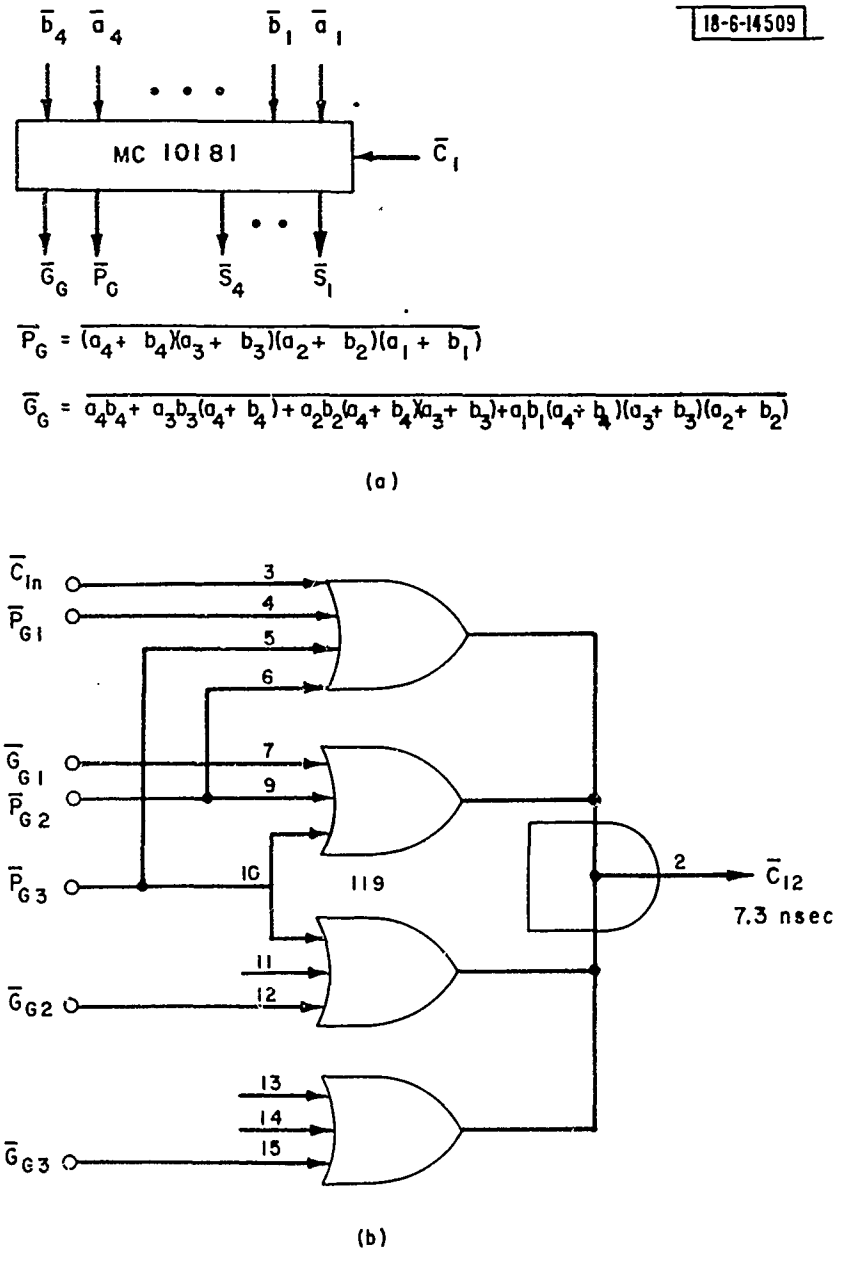


Fig. 12. Carry look ahead.

Both adders have input and output gating that adds an additional 8.3 nsec to the time it takes for data to flow through the adder function box. Thus, all 12-bit operations will take 21.8 nsec or less, and all 24-bit operations will take 29.0 nsec or less.

D. Timing

A simplified timing diagram for two consecutive add instructions (Fig. 13) will aid calculation of the time to execute 12-bit add instructions.

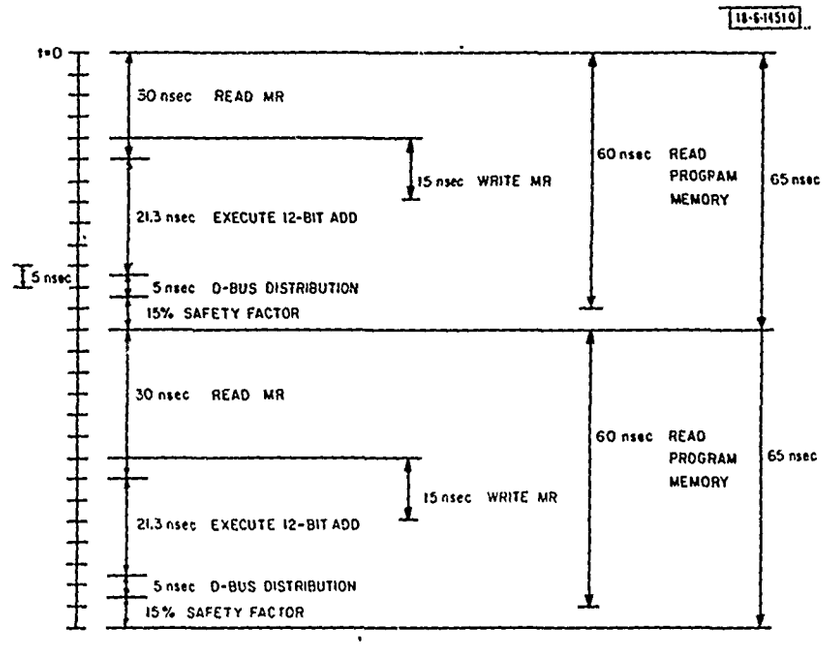


Fig. 13. 12-bit add instruction timing.

At t = 0 the first add instruction is transferred into the instruction register from the program memory, and the program address register that had previously contained the address P is incremented by 1 so that its new value is P + 1. From the section on general registers, it takes 30 nsec to read the two operands from the general registers and to transfer them to the ALU function box. It then takes 21.8 nsec to add the operands and an extra 5 nsec to send the result from the function box back to the general registers via the D bus. The sum of these three times is 56.6 nsec. A 15 percent safety factor is added to cover delay variations due to temperature changes, power supply variations, and noise giving a total of 65 nsec when rounded to the nearest 5-nsec increment.

While the first add is executed, the next instruction (the second addition) is read from address P + 1 of the program memory. A program memory read, as explained in the next section, requires 60 nsec; this is the interval beginning when a new address is clocked into the program memory address register and ending when a new instruction arrives at the input of the instruction register. The second add instruction is, therefore, available at the input to the instruction register when the first add is complete, and a new add instruction is begun by transferring the new instruction into the instruction register. Simultaneously, the 12-bit result of the first add is clocked into the DA and DB registers (Fig. 9) from which it will be written into the general registers after operands for the current add instruction are read from the general registers. It is easy to see that the second add instruction and all subsequent add instructions taken from concurrent program memory locations are executed in 65 nsec.

When executing 24-bit additions, the timing diagram (Fig. 13) remains unchanged except that the addition time changes from 21.8 to 29.0 nsec. There is a corresponding change in the 15 percent safety factor resulting in a 75-nsec, 24-bit add, instruction time.

In general, the time required to complete any of the processor' s instructions has four components: (1) 30 nsec to read the general registers; this time increment is included in all instructions even those few for which it is not required such as JPS, an unconditional jump; (2) X nsec to perform an operation in a function box; (3) 5 nsec to transmit a result from a function box back to the general registers, if this is required by the instruction; and (4) a 15 percent safety factor.

There are two exceptions to this rule: (1) when the computed instruction time is less than 65 nsec, and (2) when certain program jumps are performed.

Some instructions such as 12- and 24-bit logic function require less time than a 12-bit addition because no carry has to propagate through the adder. Other instructions such as JPS require no more than 10 or 15 nsec to execute. Unfortunately, this speed cannot be taken advantage of because

the read of the next instruction from the program memory takes 60 nsec, which is only 5 nsec less than the 65 nsec needed for a 12-bit add. The speed of these fast instructions could be increased by 5 nsec, but there is little gain in doing so.

When a jump instruction located at address P in the program memory is executed, there is the option, under the control of the instruction's α bit, and explained in the Appendix, to skip or execute the instruction located at address P + 1. This assumes, of course, that the jump instruction commands that the program branch to a location Z not equal to P + 1. If the program is going to skip the instruction following the jump, then the processor cannot use the instruction which was read out of the program memory while the jump was in progress and must wait for the new instruction located at address Z to be read. This requires at least 60 nsec; waiting 65 nsec is proposed. The net effect is that this type of jump takes an additional 65 nsec. On the other hand, if the instruction located at address P + 1 is performed, no extra time is needed.

E. 4-Quadrant Array Multiplier

The ASP will have two 4-quadrant array multipliers that may be operated separately or in parallel at the behest of the programmer. Each multiplier function box is comprised of a network of interconnected 4-bit ALUs, specifically, the Motorola MC 10181 ALU package. A given multiplier will accept two signed, 12-bit, 2's complement operands, one from the A bus (upper or lower byte) and one from the B bus (upper or lower byte). The output (product) consists of 24 bits, the two most significant of which are considered sign bits. These are always equal except when squaring the largest negative number. All 24 bits of product may be placed on the D bus, if desired. If both operands are considered integers, only bits 1-12 of the product are retrieved and placed on either the upper or lower D-bus byte, depending on the multiplier in question. If the operands are considered to be binary fractions (binary point to the right of the sign bit), then the product is considered to be a fraction with the binary point to the right of the least significant of the two sign bits. Thus bits 12 through 23 of the output are

retrieved and placedon either D-bus byte, so that the product can be considered a binary fraction represented in exactly the same fashion as the operands.

The overflow flags associated with the upper and lower D-bus bytes can be set by the multipliers depending on the destination of the opted product bits. The rules governing overfloware:

(1) An integer multiply will set the appropriate overflow flag if bits 12 through 24 of the product are not identical. This implies that the product is not representable in 12 bits.

(2) A fraction multiply will set the appropriate overflow flag if bits 23 and 24 of the product (the two nominal sign bits) are not identical.

(3) A multiply involving a 24-bit product transfer cannot set the overflow flag on the lower D-bus byte, but will set the upper byte flag if bits 23 and 24 of the product are not identical.

Overflow conditions may be tested via the overflow jump (JOV) instruction.

The operation of the multipliers is most easily visualized by understanding 2's complement number representation* where a number is defined:

$$\underline{X} = -X_s \cdot 2^{N-1} + \sum_{i=0}^{N-2} X_i \cdot 2^i \quad .$$

Here, an N-bit binary word is considered to be the sum of two polynomials, one negative and one positive. X_s, the binary coefficient of 2^{N-1} is the sign bit. The binary coefficients X_i are the rest of the bits of the word. A signed N bit by N-bit arithmetic product may be written as follows in terms of this definition:

---

* · is multiplication, + is addition, - is subtraction.

$$\underline{Z} = \underline{X} \cdot \underline{Y} = \left[-X_s \cdot 2^{N-1} + \sum_{i=0}^{N-2} X_i \cdot 2^i\right] \cdot \left[-Y_s \cdot 2^{N-1} + \sum_{j=0}^{N-2} Y_j \cdot 2^j\right] \quad (1)$$

or

$$\underline{Z} = X_s \cdot Y_s \cdot 2^{2N-2} + 2^{N-1}\left[Y_s \cdot \left(-\sum_{i=0}^{N-2} X_i \cdot 2^i\right) + X_s \cdot \left(-\sum_{j=0}^{N-2} Y_j \cdot 2^j\right)\right] + \sum_{j=0}^{N-2} \sum_{i=0}^{N-2} X_i \cdot Y_j \cdot 2^{i+j} \quad (2)$$

This expression can be further rewritten by observing a simple implication of the 2's complement definition: the sign of a given number may be changed by complementing all coefficients and then adding 1. Mathematically speaking

$$-\sum_{i=0}^{N-2} X_i \cdot 2^i = \left(\sum_{i=0}^{N-2} \overline{X}_i \cdot 2^i\right) + 1 \quad (3)$$

and

$$-\sum_{j=0}^{N-2} Y_j \cdot 2^j = \left(\sum_{j=0}^{N-2} \overline{Y}_j \cdot 2^j\right) + 1 \quad . \quad (4)$$

Using Eqs. (3) and (4) to rewrite Eq. (2)

$$Z = X_s \cdot Y_s \cdot 2^{2N-2} + 2^{N-1}\left[Y_s \cdot \sum_{i=o}^{N-2} \overline{X}_i \cdot 2^i + X_s \cdot \sum_{j=o}^{N-2} \overline{Y}_j \cdot 2^j\right] + (X_s + Y_s) \cdot 2^{N-1} + \sum_{i=0}^{N-2} \sum_{j=0}^{N-2} X_i \cdot Y_j \cdot 2^{i+j} \quad . \quad (5)$$

Four-quadrant (all sign options) multiplication thus seems to involve a series of coefficient additions with proper weighting conventions observed. Given that X_i and Y_j are binary digits, their arithmetic product is simply a logical product (AND)*

$$X_i \cdot Y_j = X_i \cap Y_j \quad .$$

---

* $A \cap B$ is logical "AND," AUB is logical inclusive "OR."

Thus Eq. (5) can be rewritten once more as

$$\underline{Z} = (X_s \cap Y_s) \cdot 2^{2N-2} + 2^{N-1} \left[Y_s \cap \sum_{i=0}^{N-2} \overline{X}_i \cdot 2^i + X_s \cap \sum_{j=0}^{N-2} \overline{Y}_j \cdot 2^j \right]$$

$$+ (X_s + Y_s) \cdot 2^{N-1} + \sum_{j=0}^{N-2} \sum_{i=0}^{N-2} (X_i \cap Y_j) \cdot 2^{i+j} \quad . \tag{6}$$

Notice that the last term of Eq. (6) can be expanded in the form

$$\sum_{j=0}^{N-2} \sum_{i=0}^{N-2} (X_i \cap Y_j) \cdot 2^{i+j} = Y_0 \cap \sum_{i=0}^{N-2} X_i \cdot 2^i + 2Y_1 \cap \sum_{i=0}^{N-2} X_i \cdot 2^i$$

$$+ \ldots\ldots\ldots + 2^{N-2} Y_{N-2} \cap \sum_{i=0}^{N-2} X_i \cdot 2^i \quad . \tag{7}$$

If the multiplicand is assumed to be the binary word

$$\underline{X} = X_s \, X_{N-2} \, X_{N-1} \cdots\cdots X_1 \, X_0 \text{ (N bits)}$$

and the multiplier is assumed to be the binary word

$$\underline{Y} = Y_s \, Y_{N-2} \, Y_{N-1} \cdots\cdots Y_1 \, Y_0 \text{ (N bits)}$$

then Eqs. (6) and (7) lead to Fig. 14. Here is illustrated the array of weighted multiplicand coefficients to be conditionally summed, depending on the multiplier coefficients. Notice how Eq. (7) is implemented in the upper 11 rows of the array. The first three terms of Eq. (6) are incorporated as the bottom rows of the array.

There are any of a number of ways to effect the actual summing of the entities in this array, some optimized for speed, others to conserve hardware. One obvious way is to explicitly form all the logical products $X_i \cap Y_j$ (called partial products) and do a straightforward addition of the resulting partial product array as it stands. Carry and sum paths can be arranged to optimize speed performance with regard to the relative carry and sum delays inherent in the adder elements used.

18-6-14511

$$(X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_0$$

$$(X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_1$$

$$(X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_2$$

$$(X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_3$$

$$(X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_4$$

$$(X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_5$$

$$(X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_6$$

$$(X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_7$$

$$X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_8$$

$$X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_9$$

$$X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0) \cap Y_{10}$$

$$(1\ \ 1\ \ \bar{X}_{10}\ \bar{X}_9\ \bar{X}_8\ \bar{X}_7\ \bar{X}_6\ \bar{X}_5\ \bar{X}_4\ \bar{X}_3\ \bar{X}_2\ \bar{X}_1\ \bar{X}_0) \cap Y_s$$

$$Y_s$$

$$(1\ \ 1\ \ \bar{Y}_{10}\ \bar{Y}_9\ \bar{Y}_8\ \bar{Y}_7\ \bar{Y}_6\ \bar{Y}_5\ \bar{Y}_4\ \bar{Y}_3\ \bar{Y}_2\ \bar{Y}_1\ \bar{Y}_0) \cap X_s$$

$$X_s$$

$$X_s \cap Y_s$$

$$Z_{23}\ Z_{22}\ Z_{21}\ Z_{20}\ Z_{19}\ Z_{18}\ Z_{17}\ Z_{16}\ Z_{15}\ Z_{14}\ Z_{13}\ Z_{12}\ Z_{11}\ Z_{10}\ Z_9\ Z_8\ Z_7\ Z_6\ Z_5\ Z_4\ Z_3\ Z_2\ Z_1\ Z_0$$

Fig. 14. Basic array of coefficients to be summed for 4-quadrant multiply.

18-6-14512

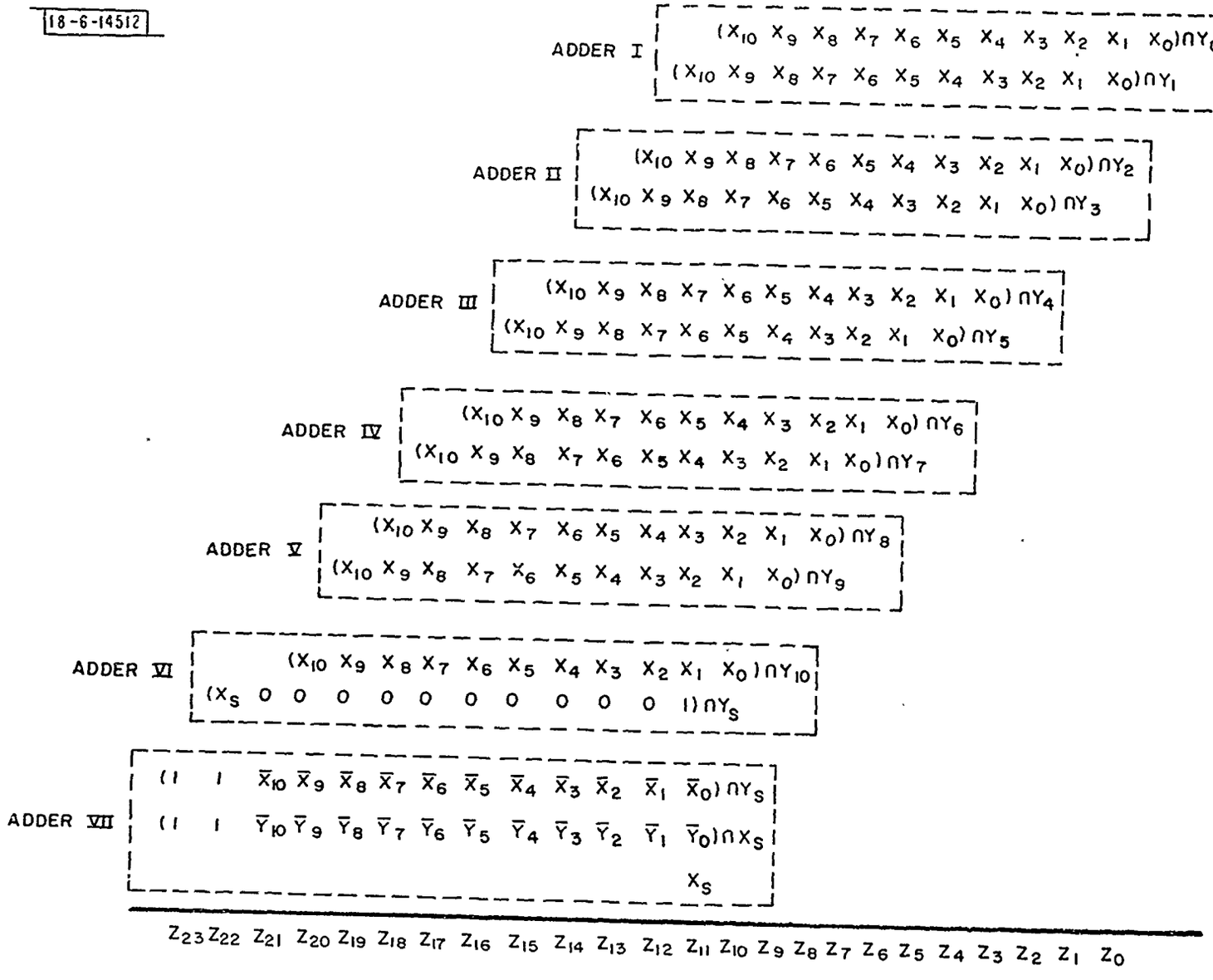


Fig. 15. Grouping of coefficient array for parallel summing.

Another method of summing the partial product array is to group the rows in pairs and add them separately, but in parallel. The results of the first stage of adds are also grouped into pairs and in turn added. The process continues until all partial sums have been combined to yield the desired product. In general, if there are N multiplier bits (including sign) the number of adder stages necessary is given by

$$S = \log_2(N+1), \text{ rounded to next highest integer,}$$

which includes the extra rows due to sign correction. For N = 12, as in the ASP, the number of stages necessary is 4. Figure 15 shows the coefficient array for the ASP case grouped for parallel summing. This method is sometimes called the "binary tree" algorithm.

Irrespective of the actual summing mechanism used for the partial product array, it should be noticed that some simplification of the rows involving $\overline{X}_i$ and $\overline{Y}_j$ can be effected. Theoretically these rows represent negative entities and thus must be assigned sign bits equal to 1. In order to incorporate them correctly into the summing operation, the sign bits must be extended as far as is necessary to derive the requisite number of product bits. The sign extension is clear in Figs. 14 and 15. When the partial product array is formed, the "south west" corner of the array appears as in Fig. 16a. Some Boolean algebraic manipulations show that the right most

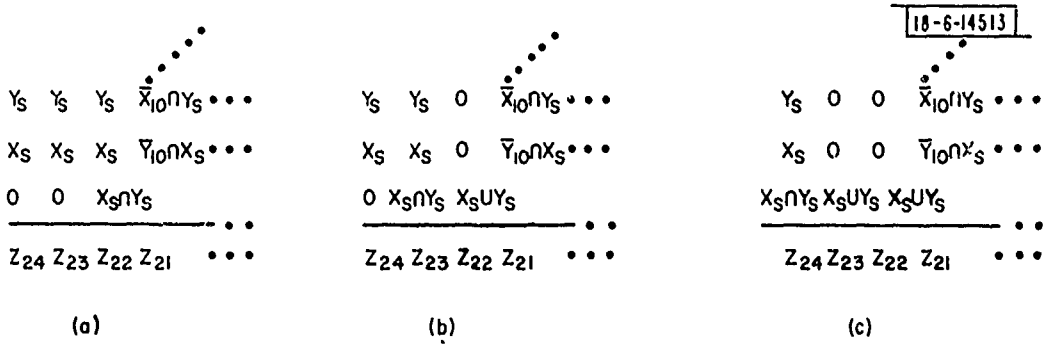


Fig. 16. Simplification of high order end of coefficient array.

column can be reduced to produce the situation shown in Fig. 16b. The center column can be similarly reduced giving rise to Fig. 16c. Clearly,

the process could be extended indefinitely. The net result is that the sign bits may be dropped off the $\overline{X}_i$ and $\overline{Y}_j$ rows if the columns containing $X_s \cap Y_s$ and beyond are replaced by $X_s \cup Y_s$. Clearly, this process need only continue as far as necessary to produce the last product bit, Z_{23} for the ASP case. The proof is as follows:

Using an adder unit, 3 bits of equal weight can be reduced to a net sum and a carry:

$$\text{SUM} = A \oplus B \oplus C_i; \quad (\oplus = \text{exclusive "OR"})$$

$$\text{CARRY} = (A \cap B) \cup (A \oplus B) \cap C_i \quad .$$

For the case at hand, let

$$X_s = A$$

$$Y_s = B$$

$$X_s \cap Y_s = C_i \quad .$$

then

$$\text{SUM} = X_s \oplus Y_s \oplus (X_s \cap Y_s) = X_s \cup Y_s$$

$$\text{CARRY} = (X_s \cap Y_s) \cup (X_s \oplus Y_s) \cap (X_s \cap Y_s) = X_s \cap Y_s \quad .$$

Therefore the sum of X_s, Y_s and $X_s \cap Y_s$ reduces to a sum equal to $X_s \cup Y_s$ and a carry into the next column equal to $X_s \cap Y_s$. The next column is now identical to the first and the process is repeated. Clearly, this can continue ad infinitum.

The actual algorithm implemented for the ASP multipliers is basically of the tree type and requires four adder stages. However, it is not necessary to explicitly form the partial product array due to the nature of the adder element used. The MC 10181 is a programmable ALU in that it can be made to perform myriad operations on the input operands in response to commands from control, or programming inputs. In the multiplier, the first stage of units is controlled by pairs of multiplier bits, the other stages are hard wired as adders. The inputs to the first stage are the multiplicand

bits, arranged for appropriate weighting. The 10181 package can be caused to add its 2 operand inputs, or gate either (or neither) through singly. These operations are all that are necessary to, in effect, form and combine the partial products.

Figure 17 illustrates the grouping of a coefficient array for a signed, 6 by 6 multiply, as an example. Three stages of adders are necessary.

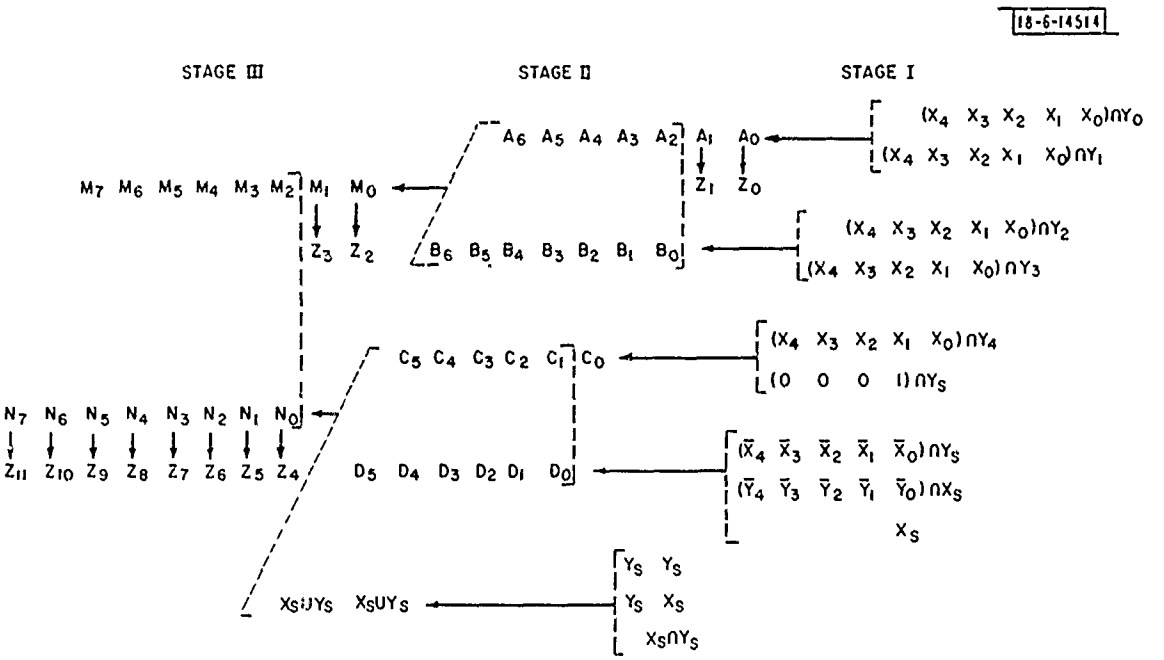


Fig. 17. Evolution of interconnections for 6 by 6 multiplier adder array.

The grouping and combining of partial sums is depicted as the process evolves from right to left. The attendant hardware realization is shown in Fig. 18. The various stages and intermediate variables labelled reference Fig. 17. Notice the manner in which the multiplicand ($\underline{X}$) is distributed to the first stage. The A input is equal to $\underline{X}$, the B input is equal to $\underline{X}$ left shifted, one place or 2 $\underline{X}$. Thus the relative weighting of subsequent rows in the array is preserved. The appropriate relative weighting of all partial sums is observed when combining them in the subsequent stages. Notice also that the control rules for each first stage unit are included in Fig. 18.

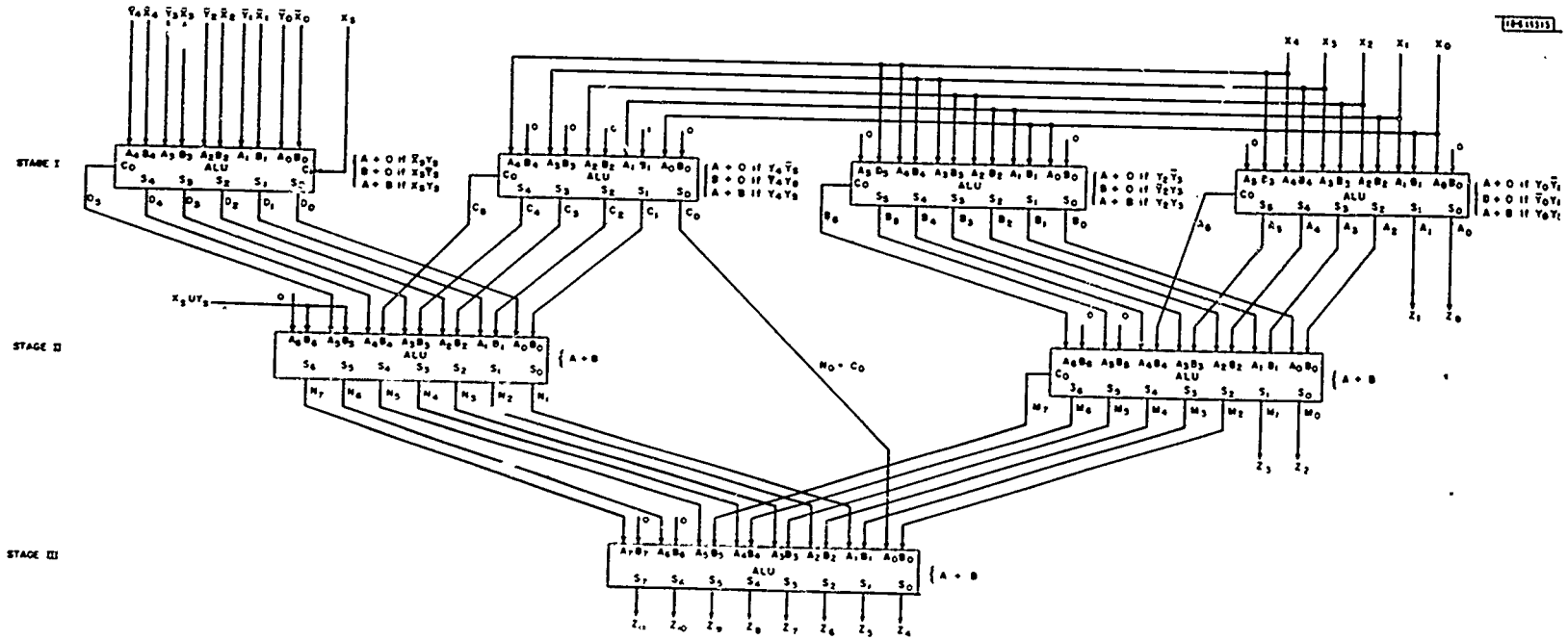


Fig. 18. Adder array for 6 by 6 multiply.

Extension to the full 12 by 12 bit case is somewhat complex, but conceptually straightforward. Figure 19 shows the basic arrangement of 10181 units for this case.

A minor modification of the array illustrated in Fig. 19 can be shown to require only 38 of the 10181 packs, and about 25 assorted 16-pin support logic packs. The basic multiplier, exclusive of control setup and operand shuffling overhead, is expected to operate in 42 nsec.

F. 4-Quadrant Array Divider

The ASP divider function box is comprised of a combinational array of adder and subtractor logic elements. The network accepts a 24-bit word from the A bus as a dividend (or numerator). The word is interpreted as a signed, 2' s complement entity with one sign bit and 23 information bits. The divisor (or denominator) is a 12-bit word that may come from either the upper or lower byte of the B bus. It is interpreted as a signed, 2' s complement number consisting of one sign bit and 11 information bits. The array produces a 12-bit quotient and a 12-bit remainder, both consisting of one sign and 11 data bits. The quotient is entered on the upper byte of the D bus, the remainder on the lower byte. The divisor and dividend may be considered to be integer, fractional, or mixed numbers. In most instances, however, it seems reasonable that the entities will be considered to be fractions with the binary point situated to the right of the sign bit. The divider overflow logic is designed to be most consistent with this interpretation.

The underlying operating principle of the array is that of nonrestoring binary division. The procedure is most easily understood by considering the divisor and dividend to be positive fractional quantities wherein the divisor is larger than or equal to the dividend. The quotient will be a positive fraction in this instance. To obtain the first quotient data bit, a trial divisor equal to half the actual divisor is subtracted from the dividend yielding a partial dividend. If the partial dividend is positive, then a 1 is entered as the quotient bit. If negative, however, the trial divisor did not "go into" the dividend and a 0 must be entered as the quotient bit. In normal (restoring) division it would be necessary at this point to add the trial divisor to the

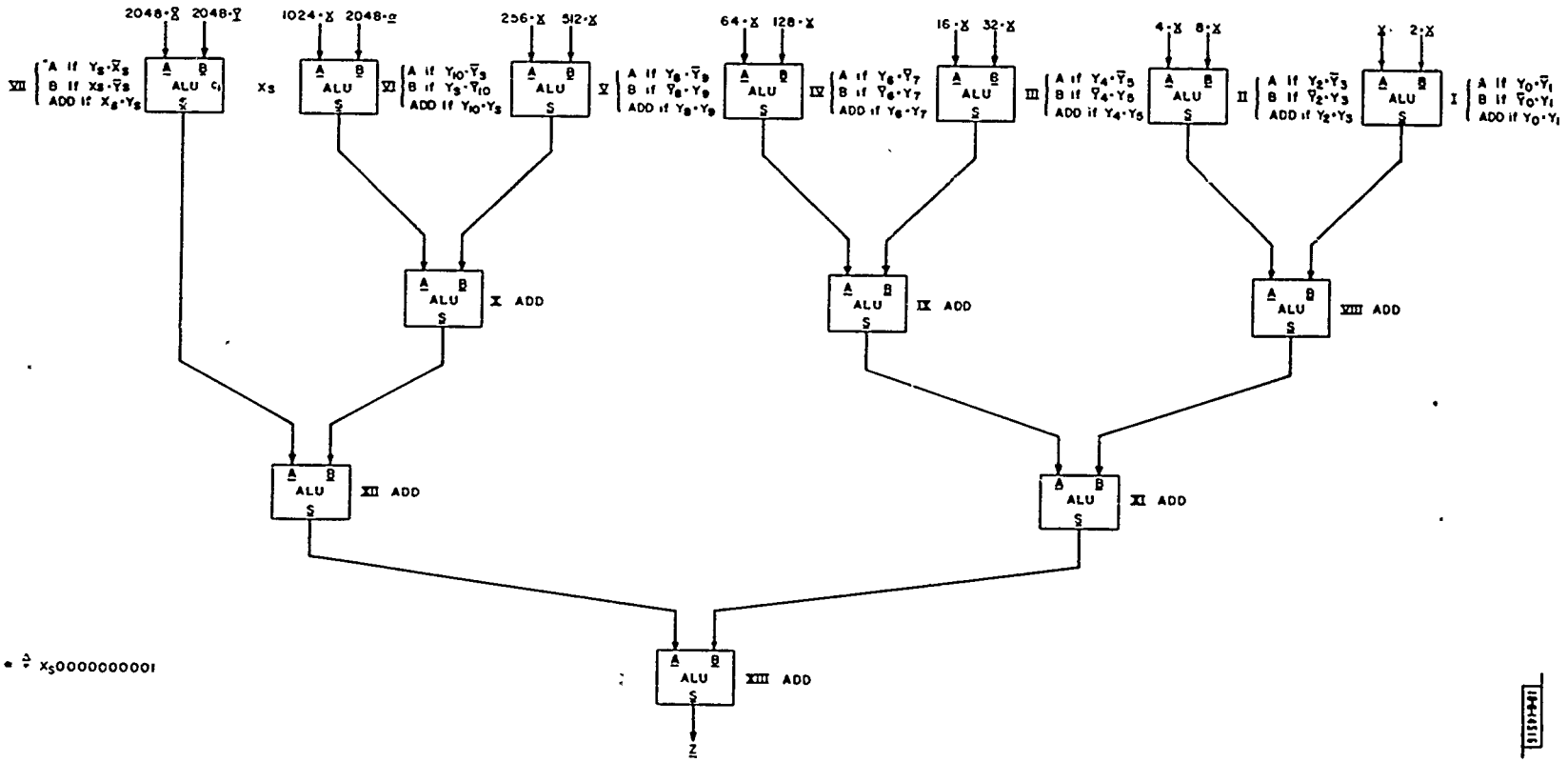


Fig. 19. Conceptual arrangement of adders to perform parallel summation of coefficient array.

partial dividend. This would restore the original dividend before an attempt is made to subtract a new trial divisor. Nonrestoring division makes use of the fact that each successive trial divisor is half the preceding one. Thus the addition (or restoration) of a trial divisor followed by subtraction of one half that very same trial divisor is nothing more than a net addition of half the trial divisor. Returning to the example, if the first data bit of the quotient is a 1, the previous trial divisor is halved and subtracted from the partial dividend to produce the next quotient bit. If the first quotient bit is a 0, the trial divisor is halved and added to the partial dividend to produce the next quotient bit. This procedure continues until all desired quotient data bits have been produced. The algorithm has the distinct advantage of being realizable as an unclocked array. There are no feedback loops; the process flows unconditionally from beginning to end without any "back-up" steps.

Realizing this division procedure in practice, for the 4-quadrant case (all sign combinations of divisor and dividend possible), requires some manipulation. The heart of the divider array consists of a series of adder/subtractor stages. Each of the stages will either add or subtract the appropriate divisor from the appropriate partial dividend depending on the sign bit of the partial dividend in question, and the sign bit of the divisor proper. The array will accept any combination of dividend and divisor signs. However, the set of quotient data bits produced by the array must be corrected at the end for certain sign combinations.

The topmost portion of a diagram for the procedure (Fig. 20) depicts generation of the quotient data bits. The bottom section depicts the end correction. The rules for generating a quotient bit at any given stage of the array are:

(1) If the present partial dividend is positive, enter a 1 as the concomitant quotient bit. If not, enter a zero.

(2) If the divisor and the present partial dividend have the same sign, subtract the next trial divisor.

(3) If the divisor and the present partial dividend have differing signs, add the next trial divisor.

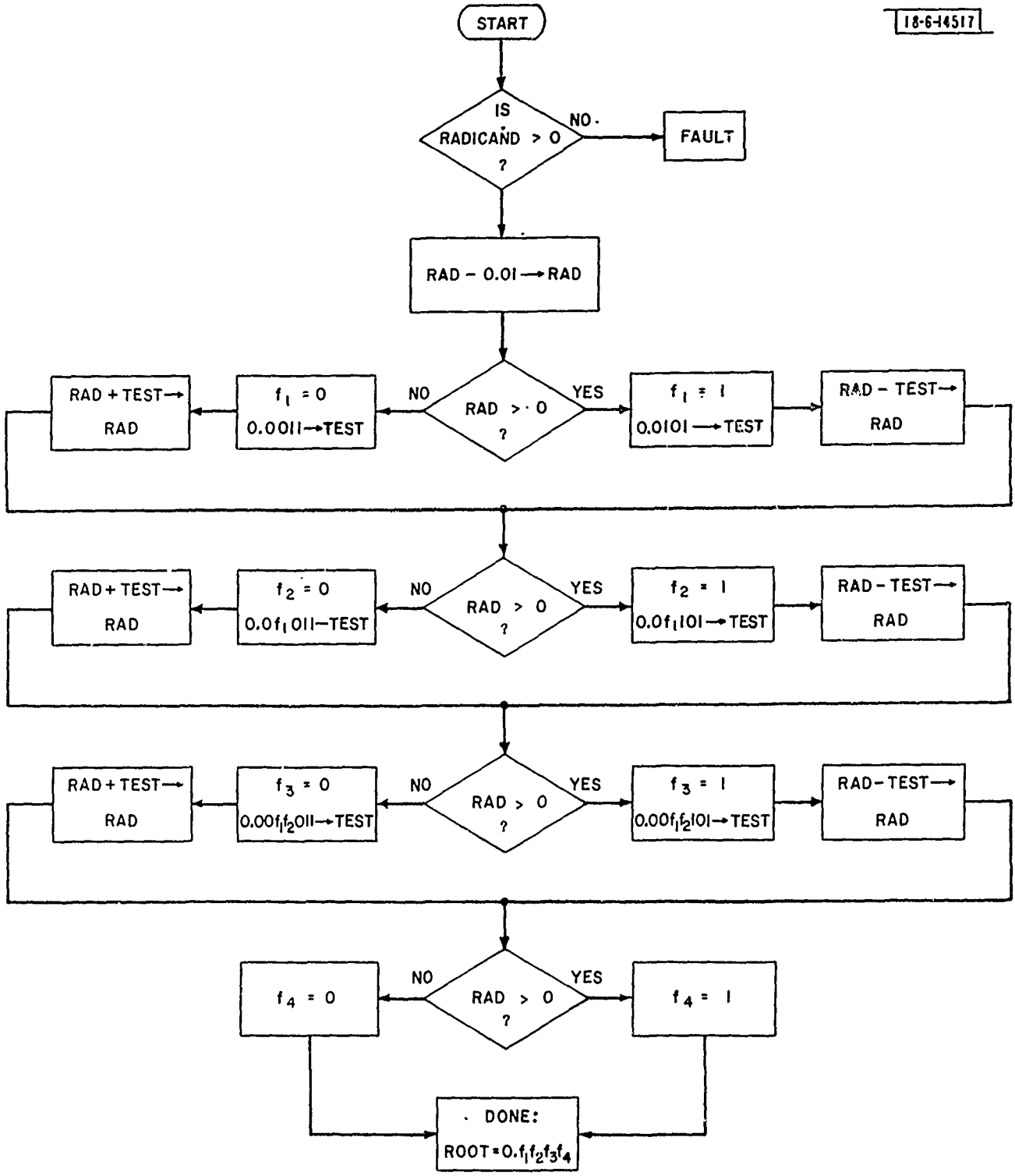


Fig. 20. Conceptual flow diagram of nonrestoring divide.

The above conventions imply two interesting facts: First, a positive dividend with either a positive or a negative divisor will yield the proper quotient magnitude. Second, a negative dividend with either a positive or a negative divisor will yield the complement of the magnitude of the quotient. A correction based on the actual signs of the operands must be applied to derive a correct 2's complement quotient representation. For example, suppose the divisor is positive and the numerator is negative. The quotient bits generated will turn out to be a 1's complement representation of the correct negative result. Thus the result must be incremented by 1 to yield the proper 2's complement representation. As a further example, suppose both the divisor and dividend are negative. Clearly the quotient ought to be positive. The array produces, however, the complement of the correct results and a pure inversion of the quotient bits is necessary. All of these cases are dealt with via an extra adder/subtractor stage at the very bottom of the array which performs the actual correction. Only one case needs no correction: positive divisor and positive dividend. The correction rules are:

(1) if both the divisor and dividend are positive, assign the quotient sign bit the value 0 and do nothing to the quotient data bits.

(2) If both the divisor and dividend are negative, assign the quotient sign bit the value 0 and complement the quotient data bits.

(3) If the divisor is positive and the dividend negative, assign the sign bit the value 1 and increment the quotient data bits by 1.

(4) If the divisor is negative and the dividend is positive, assign the sign bit the value 1. Complement the quotient data bits and increment by 1.

Conceptually, all array additions and subtractions involving an N bit signed divisor can be carried out on an N + 1 bit basis. The difference (or sum) between any given partial dividend and its trial divisor should also be representable in no more than N bits (really N - 1 bits plus sign). Thus, for this case, bits 12 and 13 of any partial dividend, being both presumably

sign bits, ought always to be in agreement. If not, an overflow indication is rendered. This indication implies, in terms of fractional operands, that the dividend was greater in magnitude than the divisor. The condition is illegal because the quotient would have to be greater than 1 and, hence, could not be represented as a signed fraction. Since the quotient appears on D_u, the overflow flag for that byte is set.

The overflow condition can also be interpreted in the context of integer operands. In such an instance an overflow will occur if the magnitude of the dividend is greater than 2^{12} times the magnitude of the divisor. In such an instance the quotient would be on the order of 2^{12} which cannot be represented in 11 data bits plus sign.

If operating with mixed operands, it would be incumbent upon the programmer to ascertain the implied overflow conditions appropriate to his own representation conventions.

Note that to generate the N-1 bit quotient and a sign from an N-1 bit divisor plus sign, only 2(N-1) bits of dividend plus a sign are necessary. This implies that the least significant bit of the 24-bit dividend operand (A-bus input), never enters the calculation of the quotient.

The partial dividend that determined the last quotient bit (i.e., the result of the last add/subtract stage) is considered to be the remainder. It is a 12-bit entity (11 bits plus sign) and is related to the other operands by the equation:

$$\text{DIVIDEND} = (\text{QUOTIENT}) \times (\text{DIVISOR}) + \text{REMAINDER} \ .$$

It can be used in conjunction with more dividend bits to derive an extended precision quotient. The actual formation of the extended dividend is involved, but it can be done and the signed remainder is necessary.

Figure 21 shows an actual hardware realization of a divider that accepts a 4-bit divisor and an 8-bit dividend yielding a 4-bit quotient and a 4-bit remainder. The realization can be extended in a straightforward manner to the 24-bit/12-bit case. The adder/subtractors represented can be realized with the MECL 10K, 4-bit, ALU package (MC 10181). The unit can be programmed to either add or subtract in response to a control. The actual

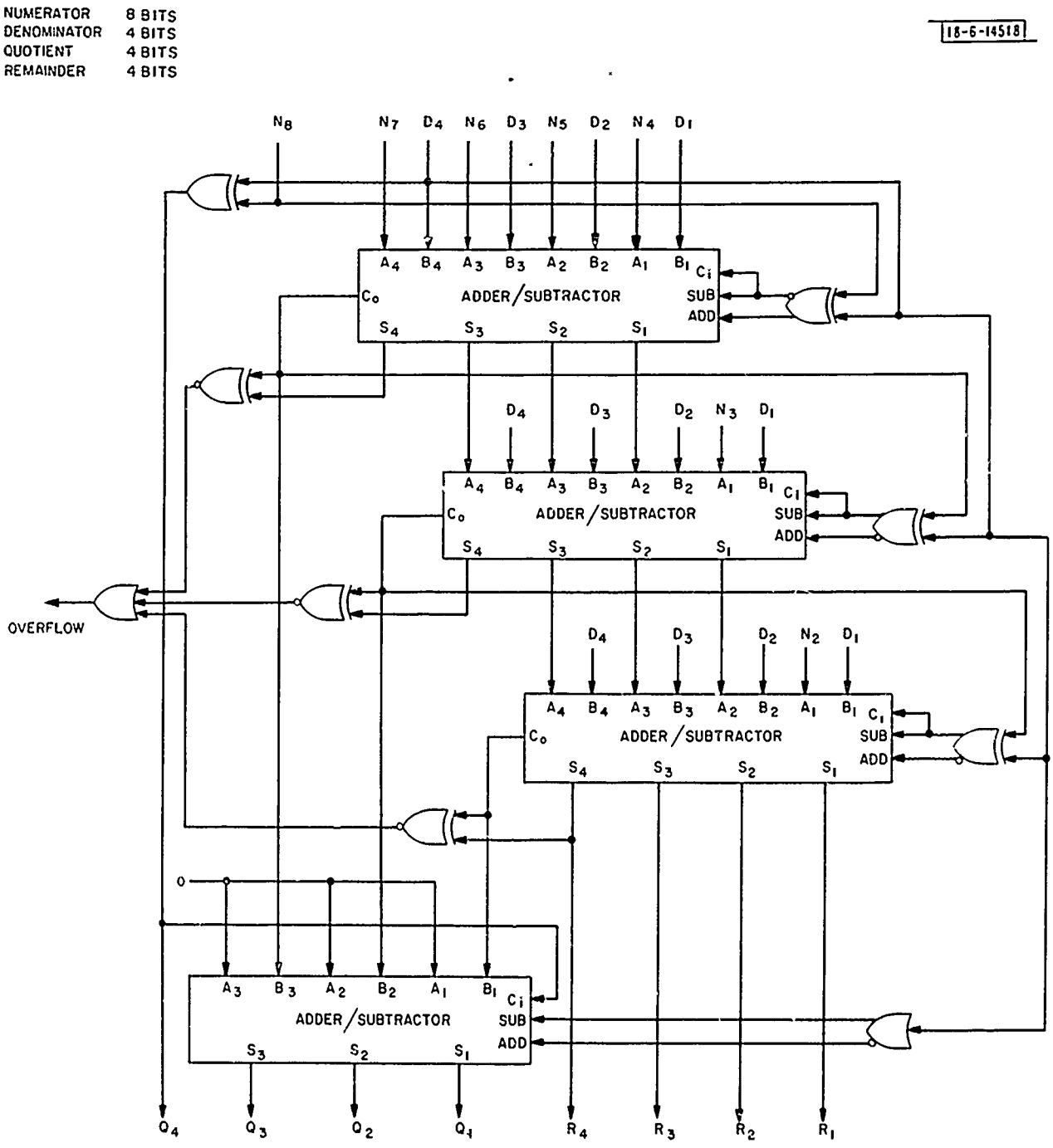


Fig. 21. 4 by 8 nonrestoring divider array with overflow detection and end sign correction.

subtraction is accomplished by changing the sign of the B input and adding. This operation, in effect, requires that the B input be inverted and incremented by 1. The package does the complementation internally but the 1 must be supplied at the C_i (carry "in" 0) input wherever a subtraction is to occur.

It might be expected that since the divisor is a 4-bit entity, all add/subtracts ought to be done on a 5-bit basis as was inferred earlier. It can be shown via some manipulation, that the N+1st bit can be simply realized as nothing more than the carry out of the Nth bit with a slight change in rules. The simplified rules now can be stated succinctly:

(1) 1st Stage - If the sign bits of the divisor and dividend differ, then add. If not, subtract.

(2) All Subsequent Stages - If the carry out (C_o) of the Nth bit of the previous adder/subtractor is a 1, enter 1 as the quotient digit. Also, if the carry out is different from the divisor sign bit, set the present stage to subtract. Add otherwise.

(3) Overflow - If the carry out of the Nth bit for any given stage is the same as the Nth bit out of that stage, signal an overflow.

(4) End Correction - can best be summarized in tabular form:

Sign of Divisor (S_D)	Sign of Dividend (S_N)	Quotient Correction	Sign of Quotient (S_Q)	Add	Subtract	Carry In (C_i)
+	+	(None)	+	✓		
+	-	Q+1	-	✓		✓
-	+	$\overline{Q}$+1	-		✓	✓
-	-	$\overline{Q}$	+		✓	

Logic equations are easily derived:

$$S_Q = C_i = S_N \oplus S_D$$

$$SUB = S_D$$

$$ADD = \overline{S}_D$$

These are seen as the controls implemented in the figure for the end correction stage. Notice that the "A" input to this stage is necessarily a hard-wired zero.

The actual 24-bit/12-bit divider is realized using 12, 12-bit stages. The first 11 derive the quotient bits, the last does the correction. Each stage requires three MC 10181 packages with a look-ahead carry generator arranged to feed bit 9. Thus 36 MC 10181 units are required. Each stage is capable of producing all necessary partial dividend bits in 13 nsec. Therefore the entire operation will require 12 x 13 = 156 nsec. The number of packages required to synthesize controls, overflow functions, and perform data distribution, is incidental. Thus, the entire unit is smaller in terms of packages than the multiplier function box. The actual net divide instruction execution time will be greater than 156 nsec due to overhead associated with control decoding, operand fetch, and deposition of the quotient.

G. Square Root Function Box

The ASP square root function box, an optional extra feature, is comprised of a combinatorial array of adder/subtractor logic in much the same manner as the divider function box. The input to the array consists of a signed, 24-bit, 2's complement number from the A bus. It is interpreted as a positive fraction, the binary point situated to the right of the sign bit. The output is a 12-bit, positive, 2's complement fraction that is placed on the upper byte of the D bus. If the input should happen to be negative, a fault condition is signalled by setting the overflow flag associated with the upper byte of D. No remainder is provided since normally more than 12 bits are necessary to properly represent it.

In analogous fashion to the divider, the square root algorithm used is one that lends itself to realization as an unclocked, combinatorial array of logic. The procedure is termed the nonrestoring square root algorithm.[3] The array is built up as a series of adder/subtractor stages. No back-up steps are required; the process moves irrevocably forward from start to finish.

To see how the procedure evolves, assume a 10-bit radicand, R, of the form:

$$R = 0.R_1R_2R_3R_4R_5R_6R_7R_8R_9 \quad .$$

The root is to be expressed as a positive fraction of the form

$$F = 0.f_1f_2f_3f_4 \quad .$$

In straightforward fashion, a series of tests can be tabulated, which should be performed on R:

(1) Is $R \geq (.1)^2$? If yes, $f_1 = 1$; otherwise $f_1 = 0$.

(2) Is $R \geq (.f_1 1)^2$? If yes, $f_2 = 1$; otherwise $f_2 = 0$.

(3) Is $R \geq (.f_1f_2 1)^2$? If yes, $f_3 = 1$; otherwise $f_3 = 0$.

(4) Is $R \geq (.f_1f_2f_3 1)^2$? If yes, $f_4 = 1$; otherwise $f_4 = 0$.

Implicit in the foregoing is the undesirable process of squaring trial radicands. By some manipulation these tests can be arranged in a more manageable form. Notice the following:

$$(.1)^2 = .01$$

$$(.f_1 1)^2 = (.f_1 + .01)^2 = .f_1{}^2 + .0001 + .0f_1 = .f_1{}^2 + .0f_1 01$$

$$(.f_1f_2 1)^2 = (.f_1f_2 + .001)^2 = (.f_1f_2)^2 + .000001 + .00f_1f_2 = (f_1f_2)^2 + .00f_1f_2 01$$

similarly

$$(.f_1f_2f_3 1)^2 = (.f_1f_2f_3)^2 + .000f_1f_2f_3 01$$

. . .

etc.

Clearly, the following tests may be substituted for the originals:

(1) Is $R \geq (.1)^2$? If yes, $f_1 = 1$; otherwise $f_1 = 0$.

(2) Is $R - (.f_1)^2 \geq .0f_101$? If yes, $f_2 = 1$; otherwise $f_2 = 0$.

(3) Is $R - (.f_1f_2)^2 \geq .00f_1f_201$? If yes, $f_3 = 1$; otherwise $f_3 = 0$.

(4) Is $R - (.f_1f_2f_3)^2 \geq .000f_1f_2f_301$? If yes, $f_4 = 1$; otherwise $f_4 = 0$.

It would appear that some squaring is still necessary. However, the squared terms can be easily formed. For simplicity, define the partial radicands and associated test values as follows:

$$R_0 = R \qquad \alpha_0 = .01$$

$$R_1 = R - (.f_1)^2 \qquad \alpha_1 = .0f_101$$

$$R_2 = R - (.f_1f_2)^2 \qquad \alpha_2 = .00f_1f_201$$

$$R_3 = R - (.f_1f_2f_3)^2 \quad , \qquad \alpha_3 = .000f_1f_2f_301 \quad .$$

Now the following set of observations can be made:

$$R_1 = \begin{cases} R, \text{ if } f_1 = 0 \\ R - .01, \text{ if } f_1 = 1 \end{cases}$$

$$R_2 = \begin{cases} R - (.f_1)^2 = R_1, \text{ if } f_2 = 0 \\ R - (.f_11)^2 = R_1 - .0f_101 = R_1 - \alpha_1, \text{ if } f_2 = 1 \end{cases}$$

$$R_3 = \begin{cases} R - (.f_1f_2)^2 = R_2, \text{ if } f_3 = 0 \\ R - (.f_1f_21)^2 = R_2 - .00f_1f_201 = R_2 - \alpha_2, \text{ if } f_3 = 1 \end{cases} \quad .$$

The pattern seems well established and the procedure for obtaining the ith root bit can be stated succinctly: subtract α_{i-1} from R_{i-1}. If the result is positive, enter $f_i = 1$. If not, enter $f_i = 0$ and add back (restore) α_{i-1}. Clearly $R_i = R_{i-1} - \alpha_{i-1}$ if $f_i = 1$, or $R_i = R_{i-1}$ if $f_i = 0$. Thus the process can continue until all desired f bits have been extracted.

Clearly, the following tests may be substituted for the originals:

(1) Is $R \geq (.1)^2$? If yes, $f_1 = 1$; otherwise $f_1 = 0$.

(2) Is $R - (.f_1)^2 \geq .0f_101$? If yes, $f_2 = 1$; otherwise $f_2 = 0$.

(3) Is $R - (.f_1f_2)^2 \geq .00f_1f_201$? If yes, $f_3 = 1$; otherwise $f_3 = 0$.

(4) Is $R - (.f_1f_2f_3)^2 \geq .000f_1f_2f_301$? If yes, $f_4 = 1$; otherwise $f_4 = 0$.

It would appear that some squaring is still necessary. However, the squared terms can be easily formed. For simplicity, define the partial radicands and associated test values as follows:

$$R_0 = R \qquad \alpha_0 = .01$$

$$R_1 = R - (.f_1)^2 \qquad \alpha_1 = .0f_101$$

$$R_2 = R - (.f_1f_2)^2 \qquad \alpha_2 = .00f_1f_201$$

$$R_3 = R - (.f_1f_2f_3)^2 \quad , \qquad \alpha_3 = .000f_1f_2f_301 \quad .$$

Now the following set of observations can be made:

$$R_1 = \begin{cases} R, \text{ if } f_1 = 0 \\ R - .01, \text{ if } f_1 = 1 \end{cases}$$

$$R_2 = \begin{cases} R - (.f_1)^2 = R_1, \text{ if } f_2 = 0 \\ R - (.f_11)^2 = R_1 - .0f_101 = R_1 - \alpha_1, \text{ if } f_2 = 1 \end{cases}$$

$$R_3 = \begin{cases} R - (.f_1f_2)^2 = R_2, \text{ if } f_3 = 0 \\ R - (.f_1f_21)^2 = R_2 - .00f_1f_201 = R_2 - \alpha_2, \text{ if } f_3 = 1 \end{cases}$$

The pattern seems well established and the procedure for obtaining the ith root bit can be stated succinctly: subtract α_{i-1} from R_{i-1}. If the result is positive, enter $f_i = 1$. If not, enter $f_i = 0$ and add back (restore) α_{i-1}. Clearly $R_i = R_{i-1} - \alpha_{i-1}$ if $f_i = 1$, or $R_i = R_{i-1}$ if $f_i = 0$. Thus the process can continue until all desired f bits have been extracted.

Step Number	If $R_i \geq 0$, $R_{i+1} = R_i - (\ \ )$	If $R_i < 0$, $R_{i+1} = R_i + (\ \ )$	Root Status
1	.01	X	0.
2	.0101	.0011	$0.f_1$
3	$.00f_1101$	$.00f_1011$	$0.f_1f_2$
4	$.000f_1f_2101$	$.000f_1f_2011$	$0.f_1f_2f_3$
5	$.0000f_1f_2f_3101$	$.0000f_1f_2f_3011$	$0.f_1f_2f_3f_4$
. . .	. . .	. . .	. . .

The process has been reduced to one of subtracts or adds of the appropriate test values (α_i). The test values to be added or subtracted at any given stage are in fact identical except for the second and third from least significant bit. Whether an add or a subtract is to occur at any given stage is wholly a function of the sign of the partial radicand (R_i) at that point.

Figure 22 depicts a conceptual flow chart of a nonrestoring realization of the example posed earlier. Figure 23 illustrates a hardware formulation based on adder/subtracter elements. The specific case shown is one of an 8-bit radicand. It should be clear that to generate N bits of root, only 2N bits of radicand are necessary. This implies, in like fashion to the division case, that the least significant bit of the radicand never enters the calculation. In the case of the ASP, only 11 bits of root (plus a sign) are necessary. Hence only the 22 bits of radicand after the binary point are used.

Figure 24 is a practical hardware realization of the case shown in Fig. 23, using the MC 10181 ALU unit. It can be shown, through detailed manipulation, that the lengths of the adder/subtracter stages can be abbreviated somewhat to conserve logic (specifically MC 10181s). In the case of the ASP realization this savings is sizable. The ASP square root function

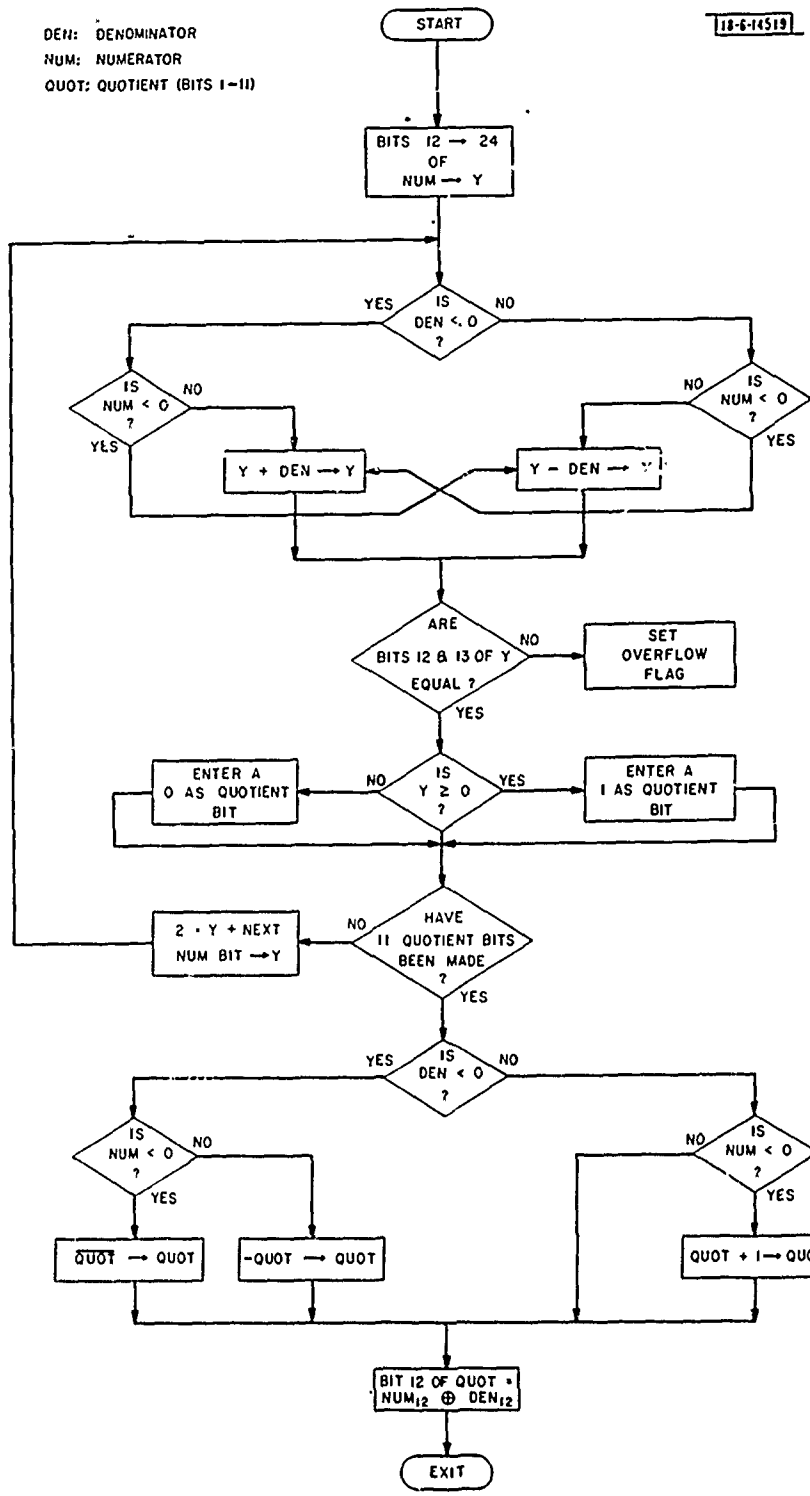


Fig. 22. Conceptual flow chart of the extraction of four bits of square root via the nonrestoring algorithm.

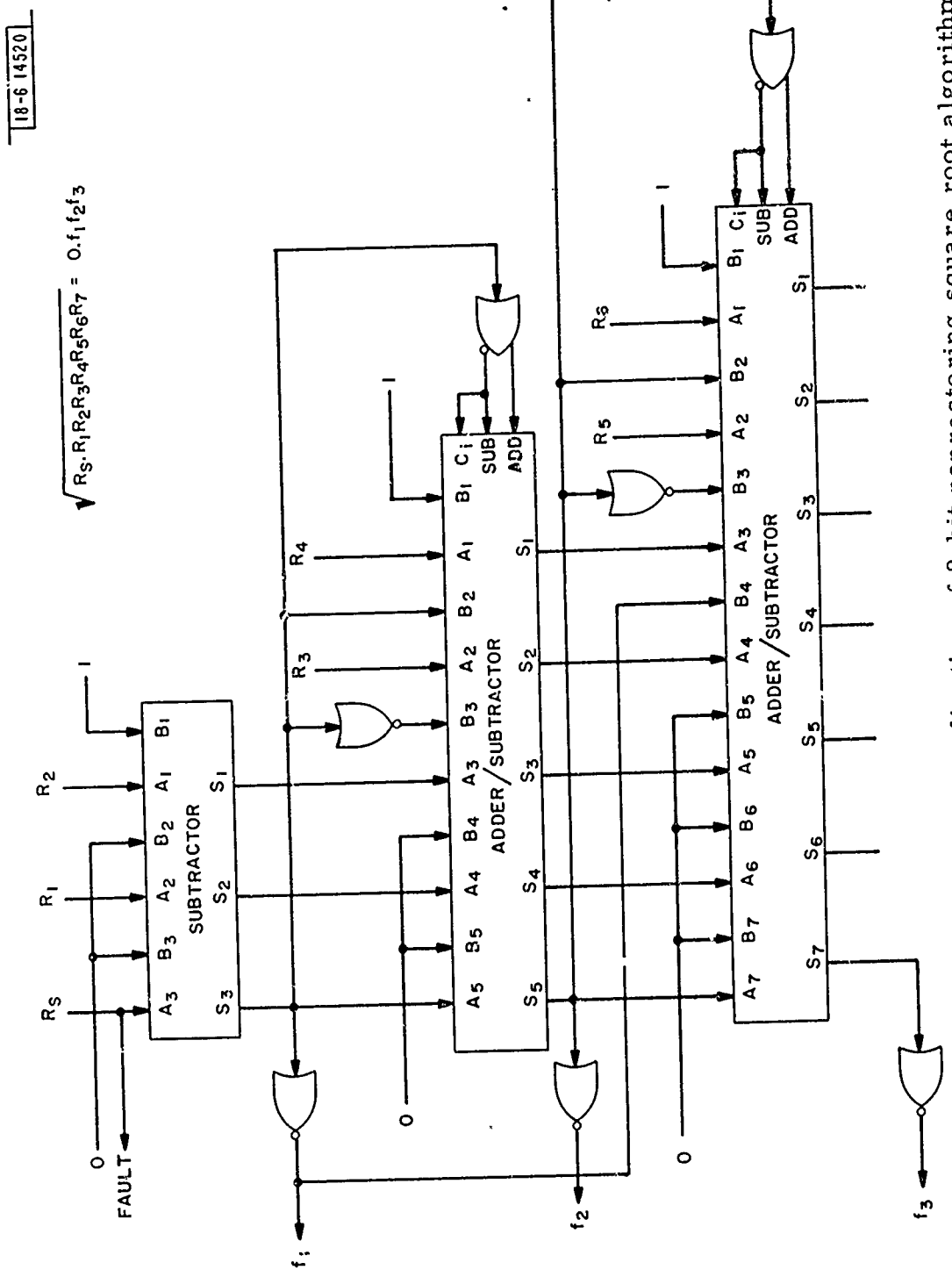


Fig. 23. Conceptual hardware realization of 8-bit nonrestoring square root algorithm.

box is realizable with 22 of the MC 10181 units, approximately half those necessary for a multiplier or a divider. The heart of the array should operate somewhere in the vicinity of 100 nsec if look-ahead carry blocks are used in the last four stages. There is, of course, the additional fixed overhead delay of operand fetch, op code setup, and the like.

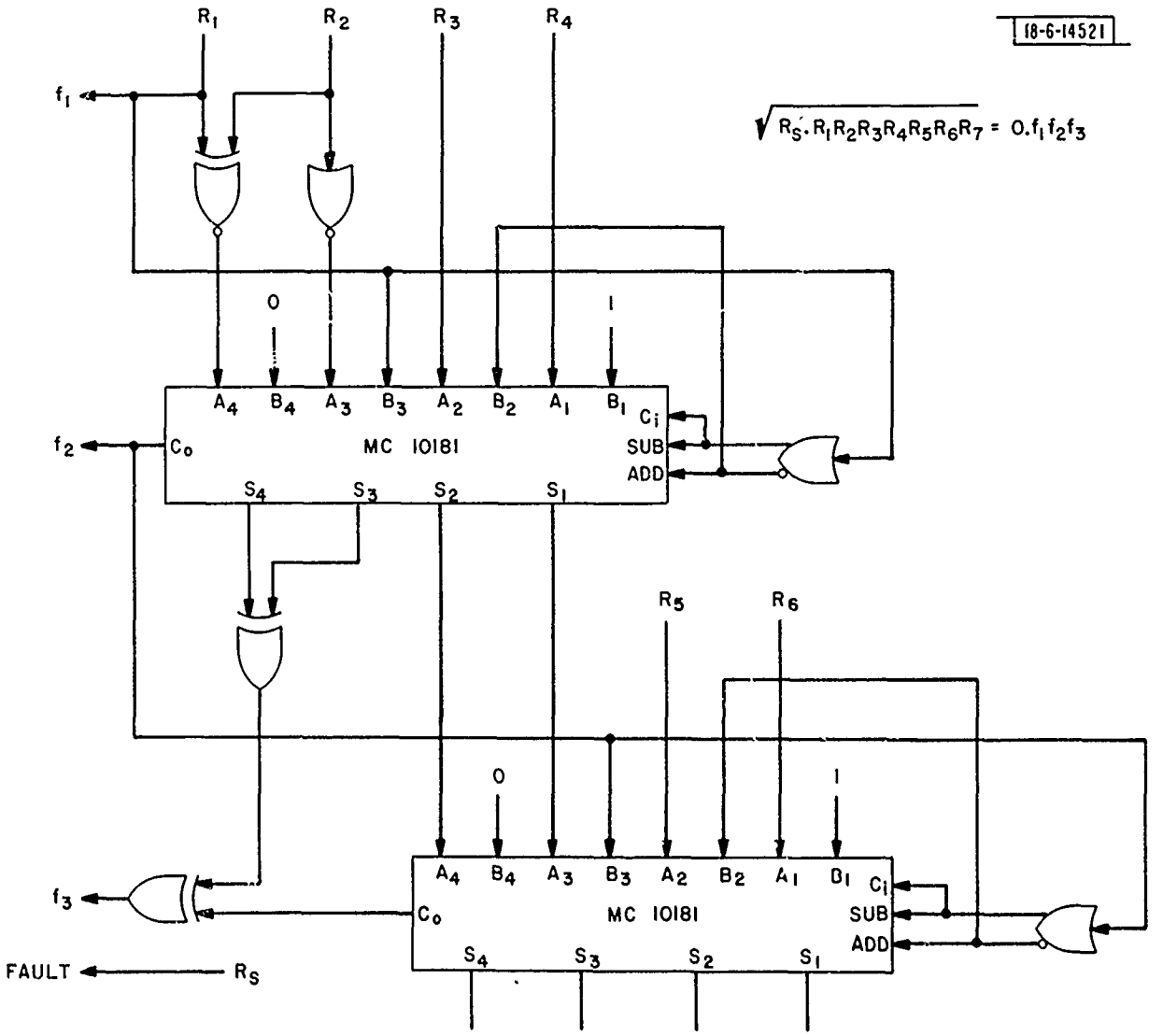


Fig. 24. Equivalent realization of 8-bit square root.

H. Special Functions

The ASP has been provided with several special function instructions to facilitate the programming of scaling operations, floating point arithmetic, double-precision multiplication, and bit-reversed counting (for FFT implementation). These features are included in the arithmetic/logic function box hardware but are sufficiently specialized to be discussed separately.

1. Bit-Reversed Add

The bit-reversed add (BRA) requires that the lower byte of a specified general register be bit-reversed (bit 1 and bit 12 interchanged, bit 2 and bit 11 interchanged, etc.) and added to the lower byte of a second general register. The carry is to propagate from bit 12 to bit 1 (left to right) and the sum replacing the contents of the lower byte of the second general register. This task is most easily effected by bit reversing the contents of the second general register, performing a normal add, and then bit reversing the sum:

$$\left[A + BRV\,(B)\right]_{\substack{\text{Carry Left}\\ \text{to Right}}} = BRV \left[BRV\,(A) + B\right]_{\substack{\text{Carry Right}\\ \text{to Left}}} .$$

The operation requires some additional gating on the inputs and output of the adder.

2. Zero Inject (ZINJ)

This operation involves simply a right shift of the contents of the lower byte of a selected general register. However, bit 12 does not recirculate as is the case in normal, signed right shifts. A zero is unconditionally shifted into bit 12. Implementation involves the normal shifting hardware with a special inhibit on the bit 12 recirculate loop. The shift is necessary to kill interference from the sign bit when combining cross products in a programmed, double-precision multiply.

3. Scale Function (SF)

The scale function operation yields a positive, 12-bit number whose magnitude is equal to one less than the number of leading 1s or 0s in the

contents of the upper byte of a selected general register. This entity corresponds to the number of left shifts that would be required to normalize the contents of the selected general register. If converting to floating point, the negative of the scale function output corresponds to the actual associated exponent of the shifted quantity. If operating in floating point, the scale function output must be subtracted from the exponent of the entity to be shifted to yield net exponent of the normalized result.

The hardware necessary to perform the SF operation is shown in Figs. 25 and 26. Figure 25 depicts a network which accepts 11 bits of input and produces a series of 10 outputs. The number of the outputs in the "true" state corresponds to the number of left shifts necessary to normalize the number ($\underline{x} = x_1 x_2 \ldots x_5$). The network of Fig. 26 is simply an interconnection of full adders (FA) to sum the number of 1s in the output of the previous network. Only four outputs are produced since the maximum number of shifts that can be required is $10 = 12_8$ which is representable in four bits. Bits 5 - 12 of the output are always zero. The hardware necessary to realize the SF operation involves only about a dozen IC packages.

4. Positive Scale Factor (SFACP)

The SFACP operation involves the transformation of a 4-bit number N, into a 12-bit number, 2^N. A subsequent integer multiply of 2^N, and the contents of a selected general register byte, will result in a net left shift of the selected quantity N places. This permits use of the multipliers in performing shift operations. The shifts might be involved as part of normalizing or straight scaling operations. For example, three steps are involved in normalizing:

(1) SF find number of left shifts necessary
(2) SFACP translate $N \rightarrow 2^N$
(3) MUL do actual N place left shift with an integer multiply.

The 4-bit input is actually the low order four bits of the appropriate 12-bit general register byte.

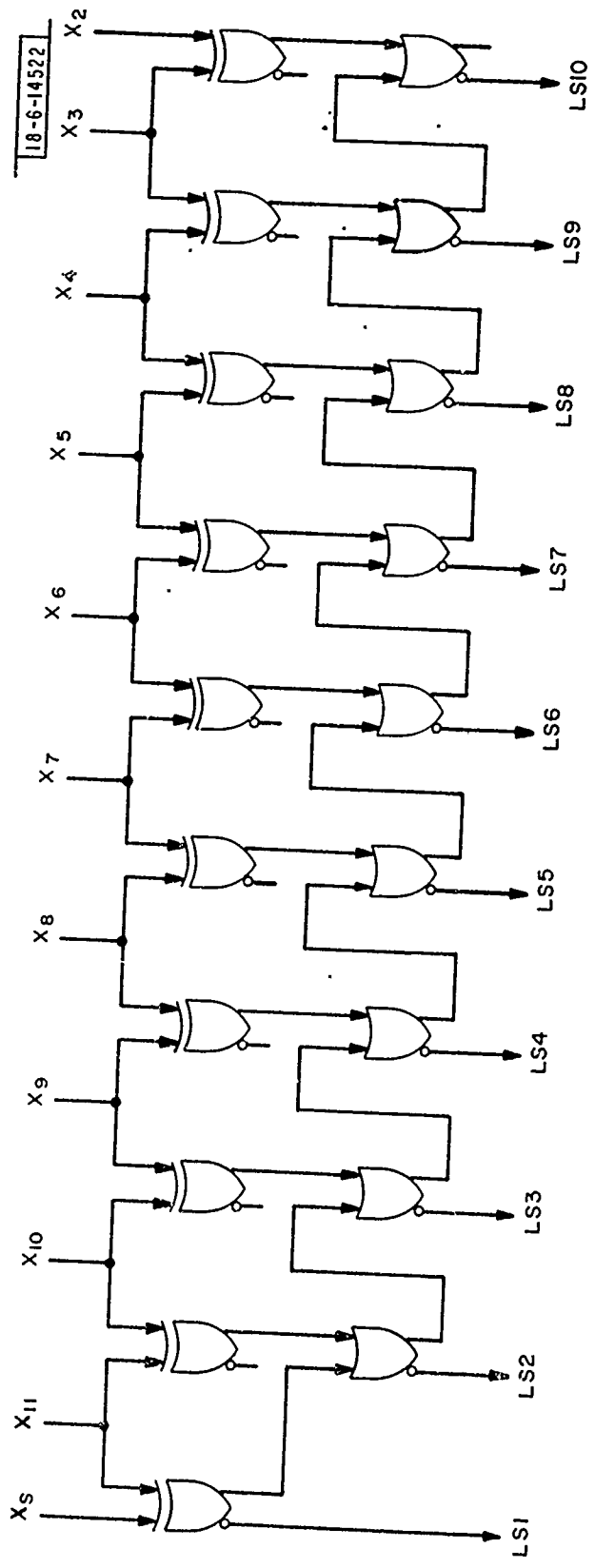


Fig. 25. Network to count the number of left shifts necessary to normalize.

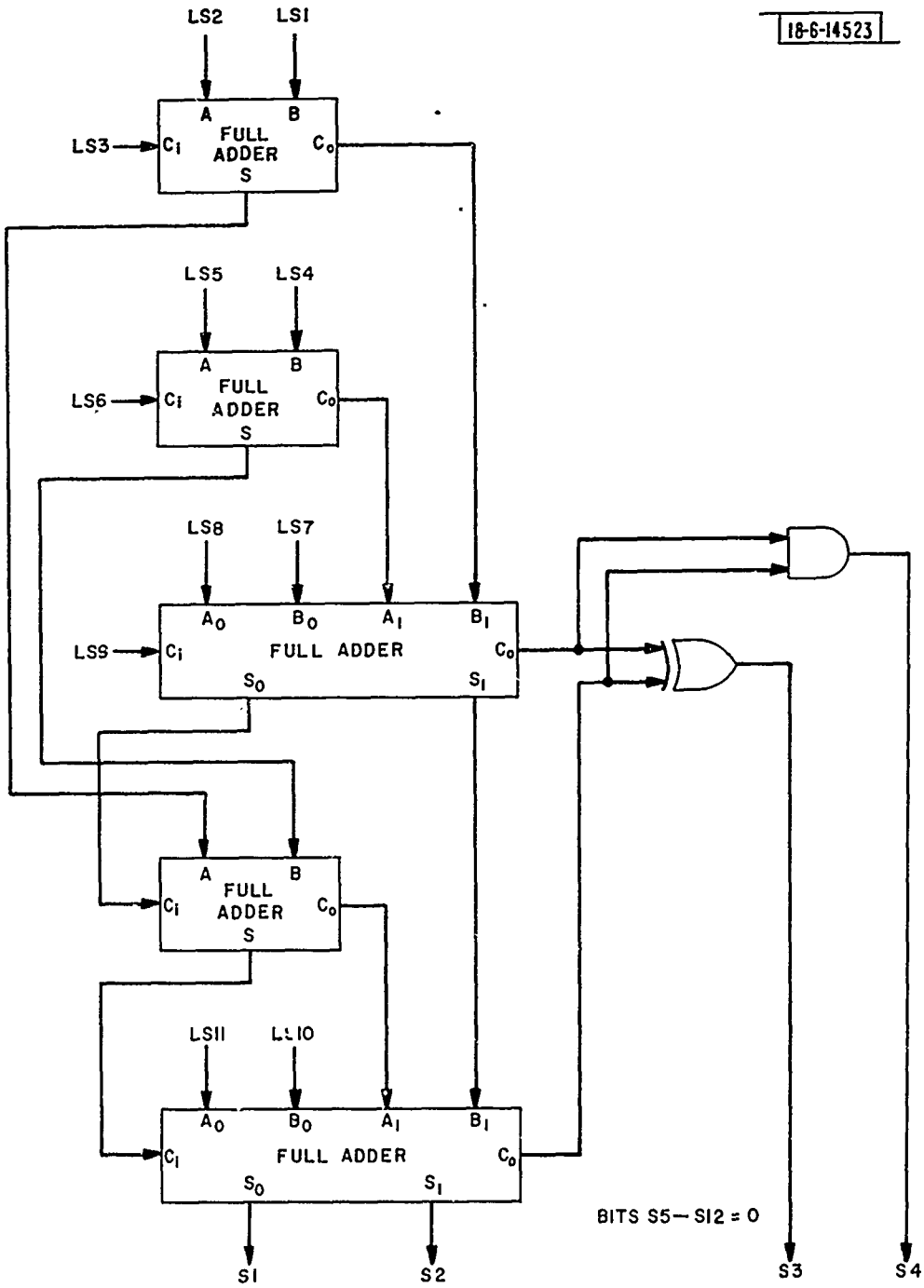


Fig. 26. Network to map left-shift count into 4-bit binary number.

The hardware realization of the SFACP operation is depicted in Fig. 27. It accepts four input bits and yields 12 output bits, the most significant of which is always a zero. Thus the multiplier can effect at most a 10-place left shift at one time. A simple 4 to 10 decoder plus a few gates are all that are necessary to realize the mapping network. The hardware is thus negligible.

5. Negative Scale Factor (SFACN)

The SFACN operation maps a 4-bit number, N, into a positive 12-bit number, 2^{11-N}, such that a subsequent fraction multiply with the contents of a selected general register byte will effectively shift those contents N places to the right. This permits the multiplier to be used as a right shifter for scaling operations. The 4-bit input, which is actually the low order 4 bits of an appropriate general register byte, may represent any integer number in the range $0 < N \leq 11_8$. Since 2^{11} cannot be represented without overflow into the sign bit, a right shift of zero places is not permitted. This implies, for instance, that in the case of coefficient alignment for floating point operations, the possibility of equal exponents must be explicitly tested.

Figure 28 illustrates a hardware realization for the SFACN operation much akin to that for SFACP. It can be realized with exactly the same hardware except that the outputs (Y) are bit reversed.

I. Scratch and Program Memories

Although the machine' s architecture has been designed to accommodate 4096-word program and scratch memories, two identical 1024-word memories, one for the program memory and one for the scratch memory, are proposed to reduce the machine' s cost. The word length for both memories will be 24 bits organized as two 12-bit bytes, and the read and write cycle times will be 30 nsec where the time measurement begins when the address signals at the input to the memory settle down.

The basic building block for the memories is the AMS 512-word, 6-bits-per-word, bipolar semiconductor card. Nine address signals are decoded to choose one of 512 words, In addition, there are six input data

The hardware realization of the SFACP operation is depicted in Fig. 27. It accepts four input bits and yields 12 output bits, the most significant of which is always a zero. Thus the multiplier can effect at most a 10-place left shift at one time. A simple 4 to 10 decoder plus a few gates are all that are necessary to realize the mapping network. The hardware is thus negligible.

5. Negative Scale Factor (SFACN)

The SFACN operation maps a 4-bit number, N, into a positive 12-bit number, 2^{11-N}, such that a subsequent fraction multiply with the contents of a selected general register byte will effectively shift those contents N places to the right. This permits the multiplier to be used as a right shifter for scaling operations. The 4-bit input, which is actually the low order 4 bits of an appropriate general register byte, may represent any integer number in the range $0 < N \leq 11_8$. Since 2^{11} cannot be represented without overflow into the sign bit, a right shift of zero places is not permitted. This implies, for instance, that in the case of coefficient alignment for floating point operations, the possibility of equal exponents must be explicitly tested.

Figure 28 illustrates a hardware realization for the SFACN operation much akin to that for SFACP. It can be realized with exactly the same hardware except that the outputs (Y) are bit reversed.

I. Scratch and Program Memories

Although the machine's architecture has been designed to accommodate 4096-word program and scratch memories, two identical 1024-word memories, one for the program memory and one for the scratch memory, are proposed to reduce the machine's cost. The word length for both memories will be 24 bits organized as two 12-bit bytes, and the read and write cycle times will be 30 nsec where the time measurement begins when the address signals at the input to the memory settle down.

The basic building block for the memories is the AMS 512-word, 6-bits-per-word, bipolar semiconductor card. Nine address signals are decoded to choose one of 512 words, In addition, there are six input data

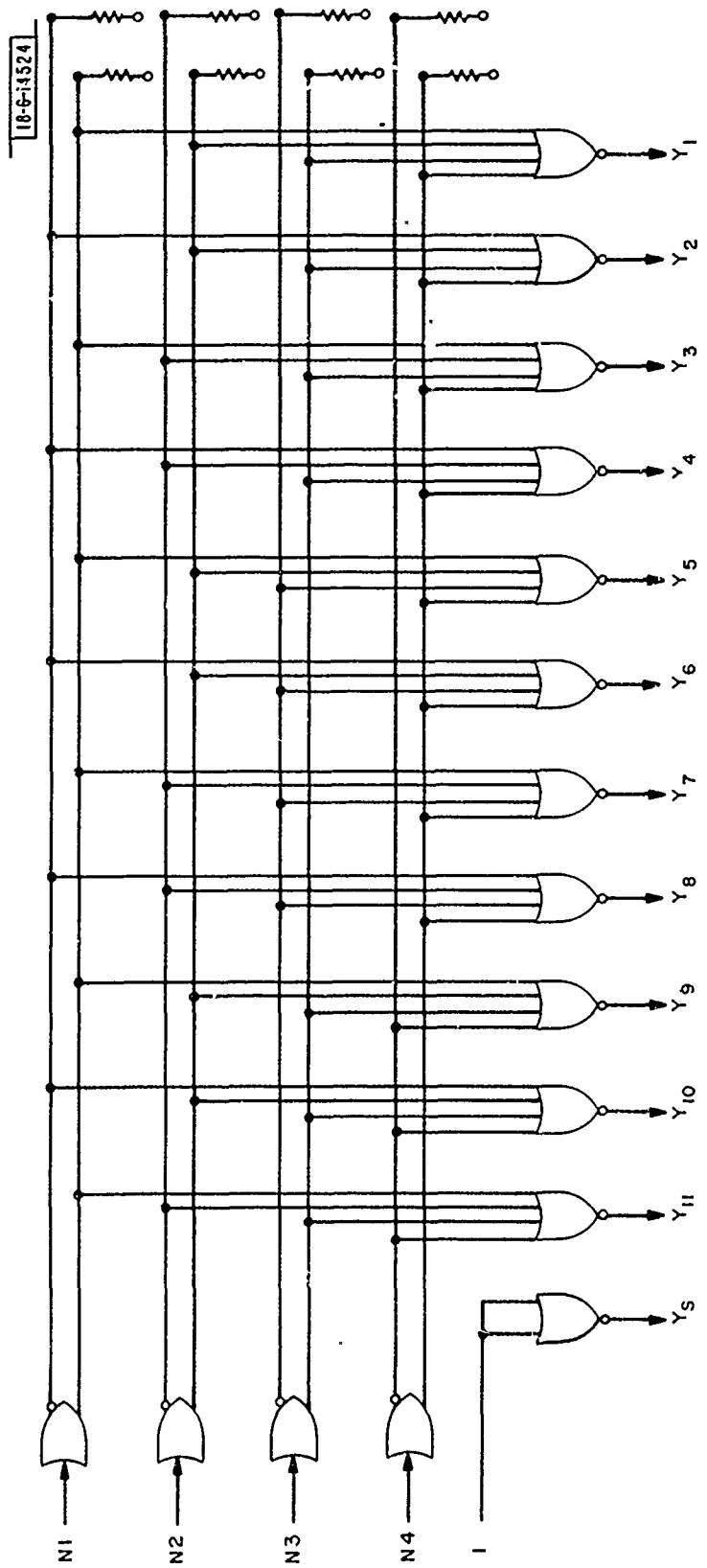


Fig. 27. Network to map N → 2^N for left shift.

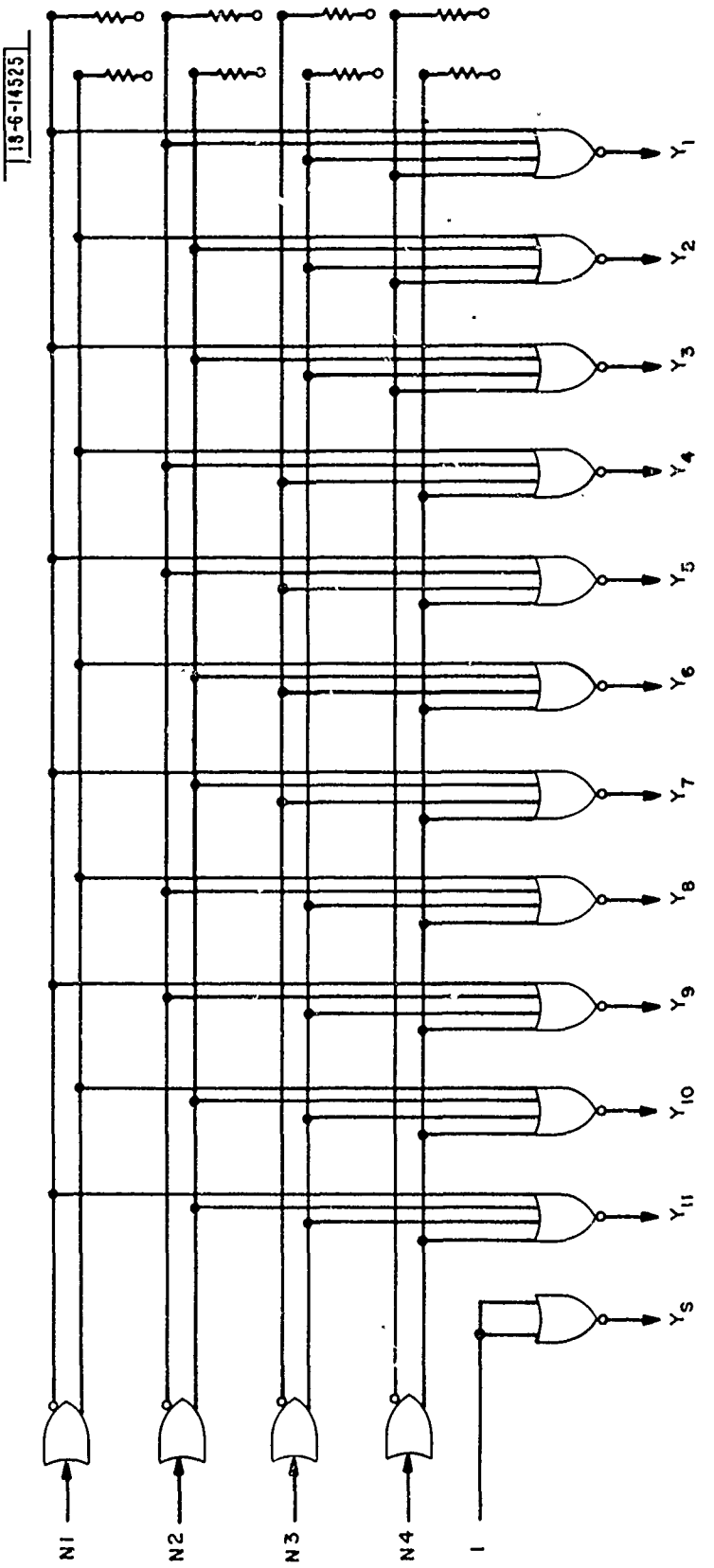


Fig. 28. Network to map $N \rightarrow 2^{11-N}$ for right shift.

signals, two card select signals, and a write signal that are sent to the card. Six data output signals are produced when the card is read. The six output signals settle down 25 nsec after the nine address signals arrive at the card if both the card select lines are low. If either select signal is high, the memory outputs will be low; thus, the select signals act as card enables and can be used to control the interconnection of memory cards to build a large memory. New data are written into the card, after the six new data signals and the address at which they are to be stored have settled down, by generating a 10-nsec write signal. The total time for the write operation is 25 nsec.

Eight AMS cards are interconnected (Fig. 29) to form a 1024 word, 24-bit memory. The cards are arranged in two groups of four cards, each

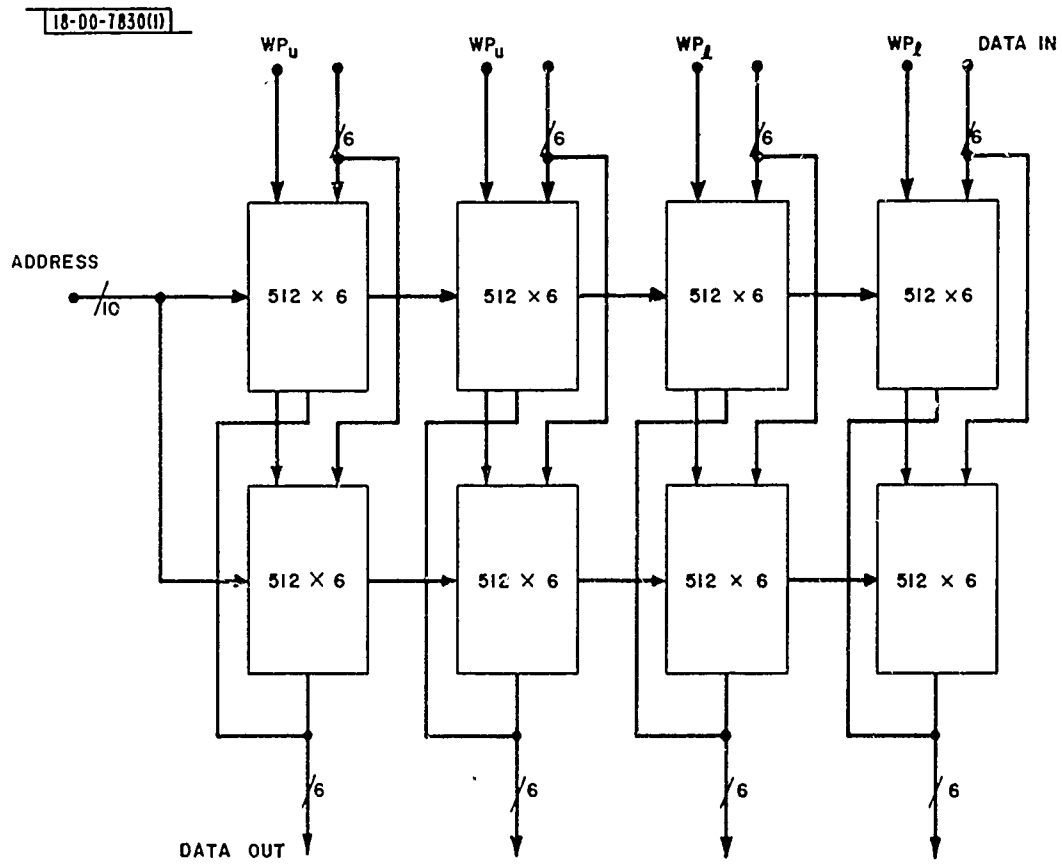


Fig. 29. Program memory/scratch memory.

group containing 512 24-bit words. Ten address signals are brought to the memory, the signals that represent the nine least significant bits of the address are sent to all eight cards. The tenth signal is used to choose which 512 word half of the memory will be read or written. Control is achieved by sending the tenth signal itself to the select inputs of one row of four cards and its complement to the select inputs of the other row of four cards. The data outputs of the two rows of cards are connected together as indicated. Once the input data and address signals have settled down, a memory write is accomplished by enabling the write signals ωp_ℓ and ωp_u. A 12-bit byte is written into the memory by forcing either ωp_ℓ or ωp_u to be true, the choice depending upon write instructions; a 24-bit word is written into the memory by enabling both ωp_u and ωp_ℓ simultaneously.

The scratch memory, M_s, for all memory instructions except for block transfers and those involving input and output from the machine, obtains its address from the B bus, its input data from the A bus, and it sends its output data to the D bus. For input-output instructions, the address and input data come from the I-O function box, and the memory output is sent to the I-O function box. On block transfer instructions, the address and input data come from the B and A buses, respectively, as they do for most other instructions, but the output data go to the program memory where it is stored, thus, giving us the capability of writing programs that modify themselves.

When a program is running, the program memory's address comes from the processor's program address register, and memory's input data come from the output of scratch memory. The current processor architecture has the output of the program memory only going to the instruction register, but the addition of a path from the program memory output to the input of the scratch memory is being considered. The path will allow us to easily check the dynamic operation of the program memory. The memory can be checked dynamically without this path, but only in an awkward manner by forcing a test program to relocate itself in the memory.

The only scratch and program memory address and data paths not mentioned are those associated with the console. These paths allow a user or another computer to specify a data word and write the word into a specific address in either memory, or to specify an address in either memory and to examine the data word stored at that address.

A scratch memory read instruction requires approximately 100 nsec and a write instruction approximately 80 nsec. The reason for the time difference is that a memory write instruction has one less data transmission path than a read instruction. When the memory is read, an address is sent from the general registers to the memory and data are returned from the memory to the general registers; whereas, when the memory is written an address and input data are sent to the memory from the general registers but no data are returned to the general registers. The 100-nsec read instruction time breaks down in the following way: 30 nsec to read an address from a general register, 10 nsec to transmit the address from the B bus to the memory assuming that the memory is in a different enclosure from the general registers, 30 nsec to read the memory, 10 nsec to send the memory output back to the D bus, 5 nsec to get the data to the general register via the D bus, and a 15-nsec safety factor. The time breakdown for a write instruction is the same except that it does not include the 15 nsec to send data back to the general registers.

It takes approximately 60 nsec to read the program memory assuming that the read time spans the interval beginning when a new instruction is clocked into the instruction register and ending when the new instruction signals arrive back at the input of the instruction register. The 60 nsec breaks down in the following way: 3 nsec for the program memory address register outputs to settle down, 10 nsec to send the address signals to the memory assuming it is in another enclosure, 30 nsec to read the memory, 10 nsec to send the memory output back to the instruction register input, and a 7-nsec safety factor.

To build a 1024-word memory, approximately six auxiliary printed circuit cards, besides the eight AMS cards will be needed. The auxiliary

cards will be used for mounting line receivers, line drivers, and address drivers. A complete memory will require approximately 600 integrated circuits and dissipate approximately 300 Watts.

J. Input-Output Capability

The ASP has eight input-output channels, two of which are full duplex, DMA data channels capable of handling 24-bit data words (Fig. 30), and are equipped with two pairs of input and output control lines. The remaining six "control" channels have no data handling capacity, but are equipped with the same control facilities as the DMA channels.

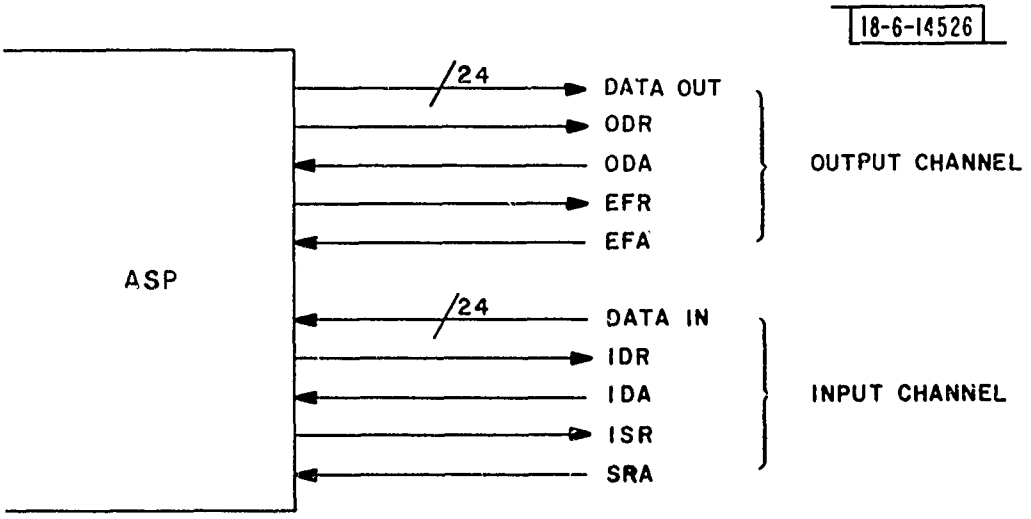


Fig. 30. Direct memory access channel.

The input side of one of the two DMA channels (channels 6 and 7) consist of 24 lines of data input, two control lines, and two acknowledge lines. Controls are named:

IDR input data request
IDA input data acknowledge
ISR input status request
SRA status request acknowledge.

If an input is desired, the ASP raises either the IDR line or the ISR, which

are presumably attached to the addressed peripheral. When the requested data are on the input lines, the peripheral raises the appropriate acknowledge line and the ASP samples the data. IDR and ISR are logically equivalent controls that provide extra flexibility: the addressed peripheral might place different types of data on the lines depending on which control line is raised. If the peripheral sees IDR, it will place a piece of data to be processed on the lines. If it sees an ISR, it will place a data word on the lines relative to its present operating condition.

Similarly, the output side of a DMA channel is equipped with 24 lines of data output, two control lines, and two acknowledge lines. Controls are named:

ODR	output data request
ODA	output data acknowledge
EFR	external function request
EFA	external function acknowledge.

If the ASP desires to output a data word, it raises either the ODR or EFR lines. When the addressed peripheral has sampled the lines, it raises the appropriate acknowledge line. ODR and EFR are logically equivalent signals. EFR might signal the addressed peripheral to interpret the incoming datum as a control word intended to establish an operating mode rather than as a simple piece of information.

These channels are termed DMA in the sense that, once a data buffer has been initiated by an appropriate program instruction, i.e., control parameters passed from M_r to the I-O handling logic, the channel automatically accesses M_s as necessary, calculates M_s addresses automatically, and signals the control processor when the entire buffer has been transmitted. The "done" signal can engender an interrupt to the user program, or can simply set a flag that can be explicitly tested by programmed instructions at the option of the user.

Input and output buffers may be active on both sides of a given DMA channel, simultaneously. It is also possible for both DMA channels to be active simultaneously with input, output, or both. In such instances,

conflicts may arise between channels for access to M_s. In fact, the program running in the CPU may also desire use of M_s at any given point. When conflicts arise between channels, M_s access will be apportioned such that channel 6 is given priority over channel 7. When conflicts arise between the input and output sides of the same channel, access will be interleaved, input being served first. In conflicts with the CPU, the CPU will be permitted to finish the instruction in progress. As soon as the CPU is finished with M_s, the highest priority I-O commitment outstanding will be serviced. Any subsequent CPU M_s accesses will be deferred until the queue of pending I-O related accesses has been processed.

The six control channels (0 through 5) (Fig. 31) are basically identical in terms of control lines to the DMA channels. The essential difference is

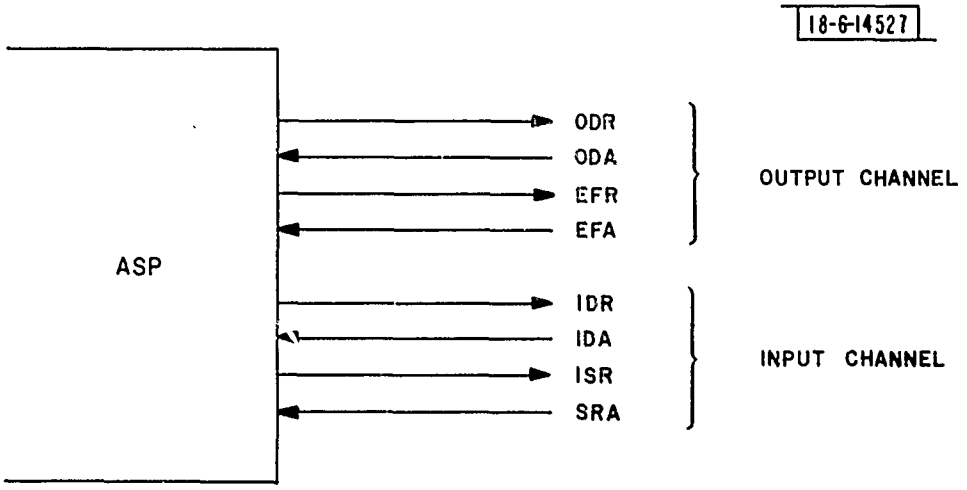


Fig. 31. Control channel.

in the absence of data handling paths eliminating the need to access M_s. This simplification greatly reduces the hardware necessary to realize a given channel. Except for the lack of data, the control channels operate the same way as the DMA channels, even to the point of interrupt generation, if desired. These channels are designed for use in computer networks where synchronization and simple control paths between processors are of value, but full duplex data links are an unnecessary luxury.

Priority issues with regard to M_s access, clearly do not arise with the control channels. However, two channels programmed to interrupt when they have completed their assigned tasks may try to signal the CPU at the same time. This situation can occur with the DMA channels, too. In such cases, the input side of a particular channel is given priority over the output side; and channel priority is determined by the channel number, i.e., the lower the channel number, the higher its priority. This implies that the control channels have priority over the DMA channels.

The 3-bit σ field in the instruction format to program the I-O system (Fig. 32a) selects the channel to be actuated. The μ (or "monitor") bit, if set, causes an interrupt to be issued when the selected channel has finished its assignment. The interrupt will cause a program branch to a prescribed subroutine. If μ is not a 1, a flag will be set when the channel is done, which can be explicitly tested. The γ field specifies the nature of the operation to be performed and is interpreted as follows:

0 input request
1 input status request
2 output request
3 external function request.

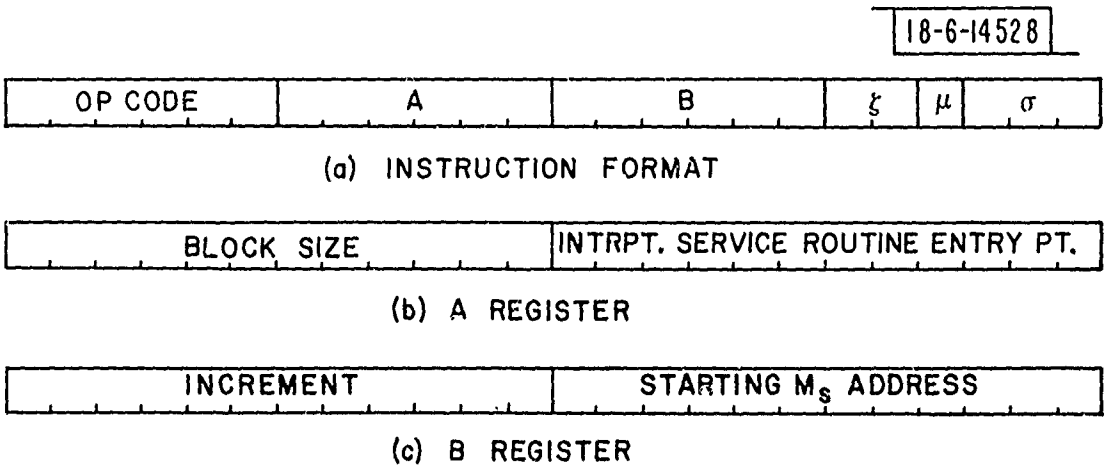


Fig. 32. I-O format conventions, (a) instruction format, (b) A register, (c) B register.

The A and B fields specify general registers that are interpreted as shown in Figs. 33b and c, respectively. These registers supply the necessary

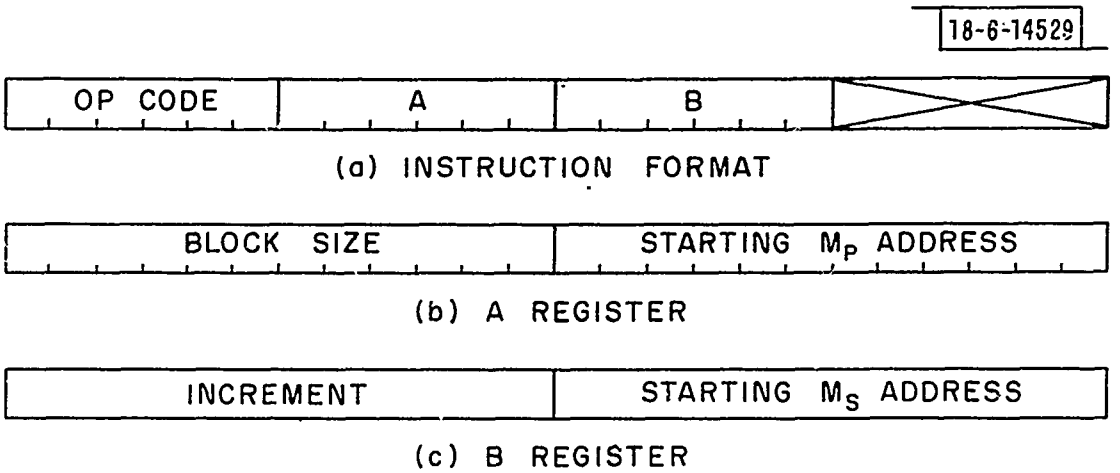


Fig. 33. Block transfer format conventions, (a) instruction format, (b) A register, (c) B register.

control parameters to the I-O logic to effect the desired task. The upper byte of A contains a number corresponding to the number of data words to be transferred. The lower byte points to the M_p location to which program control is to be transferred when the channel is done (if $\mu = 1$). The upper byte of B contains a signed number that defines the displacement between locations successively accessed in M_s. For example, if equal to +1, successive M_s locations will be accessed in order of increasing address. The lower byte of B points to the M_s location to be accessed by the first transfer. Implied by the foregoing is that only one side of one channel may be activated by a given instruction. Also, in the case of control channels, the B register is irrelevant since no data are actually transferred.

When an I- O precipitated interrupt occurs, the return point (P +2) is written into the lower byte of the A register after the service routine entrance point has been read into P. This technique saves on the number of general registers necessary to service the I-O, but requires that the lower byte of A be restored in some fashion when the service routine terminates. The RJP instruction was designed with this purpose in mind and is

documented in the Appendix along with the I-O jumps intended to explicitly test for completed I-O transactions.

Figures 33 b and c show, respectively, the instruction format and the two control parameter register formats for the block transfer instruction. The block transfer is not an I-O operation in the strict sense, but rather involves an internal two-way data transfer path between M_s and M_p. The control parameter register formats are similar to those of genuine I-O operations (the lower A byte designates the first M_p location to be accessed) as is the necessary hardware.

No interrupts are involved with block transfers. Program execution essentially halts while the transfer is in progress and resumes upon buffer completion. If the block transfer modifies M_p, the first instruction executed on resumption of normal operation is that to which control normally would have been transferred prior to the M_p modification. If no M_p modification occurred, no question arises. This instruction permits dynamic alteration of the running program and facilitates maintenance of the program memory (Appendix).

K. User Console

The ASP is equipped with a console to monitor and control the operating status of the machine, interact with and debug user programs, and to supplement standard engineering maintenance. The ASP console design is consistent with these ground rules:

(1) Preservation of Machine Status

Machine status interrogations will not alter the state of the computer: Examination of the contents of a selected M_s location will not alter the contents of the M_s address register. The same is true of data entry. The only permissible status change is that engendered by the deposition of inputted data.

(2) Complete Examination of Machine Status

Every possible useful register or group of registers viewable, and where applicable, alterable.

(3) Minimum Interconnections

Minimum signal paths connect the console and the ASP. This restriction simplifies the console, enhances cable and connector reliability, reduces required connector parts, and diminishes noise pickup that might be injected into the ASP. Noise and cable complexity are important if the console is remote from the computer. If so, a modem set for data transmission might be desirable. The amount of multiplexing required will have been drastically reduced at the outset.

(4) Possibility of Automatic Control

To permit the user to interface with the ASP via a general-purpose computer and associated I-O devices, a computer (possibly mini-computer) whose resident software can be written to simulate the presence of the console, might be installed. The monitor software could perform powerful sequences of console operations at high speed:

(a) The entire state of the ASP could be dumped on command,

(b) An entire buffer could be dumped into M_p or M_s,

(c) A particular program loop could be executed a prescribed number of times.

The possibilities of such a scheme are legion.

Tentative inputs and outputs for the console include:

I. Indicator Outputs

A. General

1.	Instruction register	IR
2.	Program memory	M_p
3.	Program counter	P
4.	Scratch memory	M_s
5.	Scratch memory address	M_sAR
6.	General register memory	M_r
7.	Bus address registers	A, B, D

B. Control

1. Machine stop
2. Machine run
3. Timing generator status

C. I-O

1. Direct memory access channels

a.	Request status	IR, OR, ISR, EFR
b.	Data input M_s address	
c.	Data input increment	
d.	Data input	
e.	Data output M_s address	
f.	Data output increment	
g.	Data output	

2. Inter-computer channels

a.	Request status	IR, OR, ISR, EFR

II. Switch Inputs

A. General

1.	Program memory	toggle
2.	Program counter	toggle
3.	Write program memory	push button
4.	Read program memory	push button
5.	Scratch memory	toggle
6.	Scratch memory address	toggle
7.	Write scratch memory	push button
8.	Read scratch memory	push button
9.	General register address	toggle
10.	Read general register	push button

B. Control

1.	Stop machine	push button
2.	Cycle machine	push button
3.	Step machine	push button
4.	Resume execution	push button
5.	Start execution at program counter switches	push button
6.	Stop when program counter equals switches	toggle
7.	Programmed stop switches	toggle
8.	Programmed skip switches	toggle

The console consists of several light registers, switch registers, and a command keyboard. The light registers permit continuous monitoring of certain machine conditions (P register, machine run/stop, timing generator status) and provide optional interrogation of others. Internal conditions may be observed on command via a general light register.

The several toggle switch registers are necessary to permit inputting two or more pieces of information simultaneously, such as address and data to load one of the memories. Some switches must also be available for continuous use: the program skip and program stop switches.

Commands are issued to the ASP via the keyboard. All command push buttons are realized this way as well as are the status interrogation options. A function code is transmitted to the ASP when each key is depressed. The code causes the console multiplexing logic internal to the ASP to bring the data desired onto the console lines. Keys corresponding to command push buttons dispatch appropriately timed pulses along with the function code to the ASP. The pulses are properly steered inside the ASP to effect the desired exercise.

L. Construction

The dashed line in Fig. 34 indicates where the system will be partitioned. The function boxes to the left of the line will be housed in a single drawer called the processor drawer and those to the right of the line will be housed in a second drawer called the memory drawer.

The circuits in the processor drawer will be mounted on either printed circuit or wire-wrap boards which are 7 in. wide and 17 in. long and have PC edge connector contacts for plugging in and out of back plane connectors.

The wire-wrap boards will have a ground plane, a voltage plane and a terminating voltage plane and will use short 2-wrap pins for better card packing density. Two types of wire-wrap boards will be used, one will hold a mix of 96 16- and 24-pin dual-in-line integrated circuits, and the other will hold approximately 130 16-pin dual-in-line circuits.

The complete processor drawer will contain approximately 1400 integrated circuits mounted on 16 boards and will dissipate approximately 300 Watts.

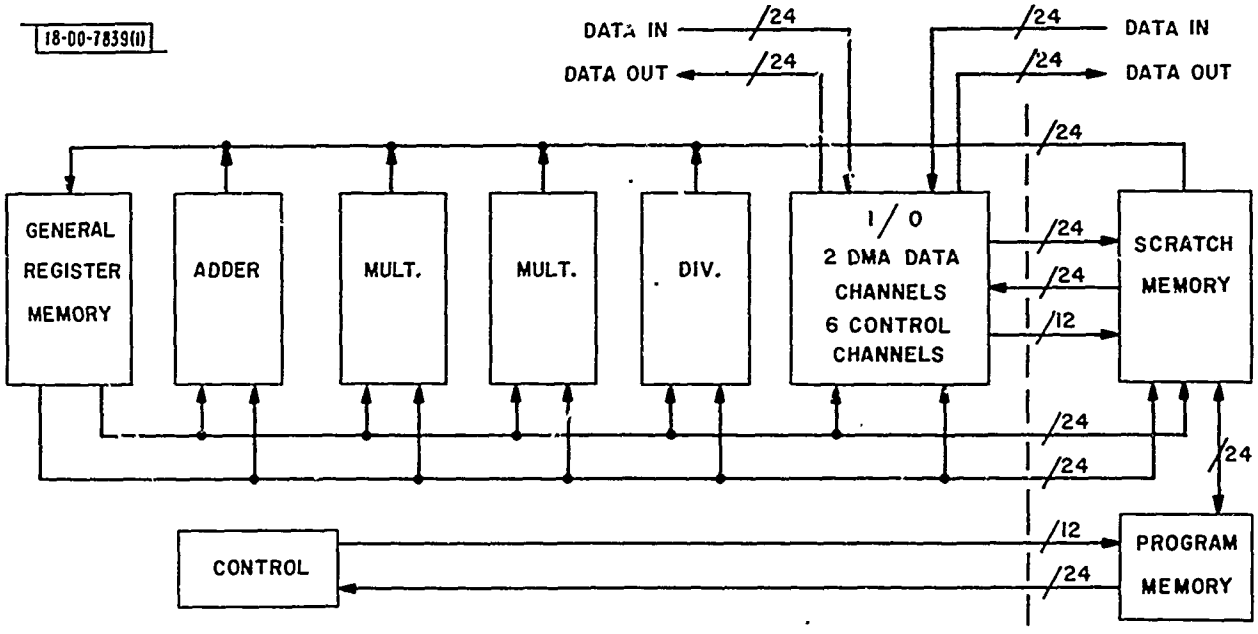


Fig. 34. System partitioning.

The memory drawer will contain the scratch memory and the program memory. There will be 16 AMS memory cards in the drawer plus another 12 auxiliary cards. Approximately 1200 integrated circuits will be mounted on the 28 cards and they will require 600 Watts.

Figure 35 shows a tentative outline drawing of the computer which reflects the desire to package the 16 processor cards in a 17 x 19 x 10 in. enclosure, and the 28 memory cards in a memory drawer that is 17 x 19 x 12 in. Cool air will be forced through both drawers to insure a maximum temperature rise of no more than 15° C. This permits operation in a 45°C ambient, which is 10°C below the 70°C maximum operating temperature of the AMS card logic and 15°C below the 75°C maximum operating temperature of MECL 10K logic.

The processor and memory drawers will be built as black boxes with few, if any, external controls. A portable console will be provided to troubleshoot programs and hardware.

The system will be powered by a bank of low voltage, high current supplies.

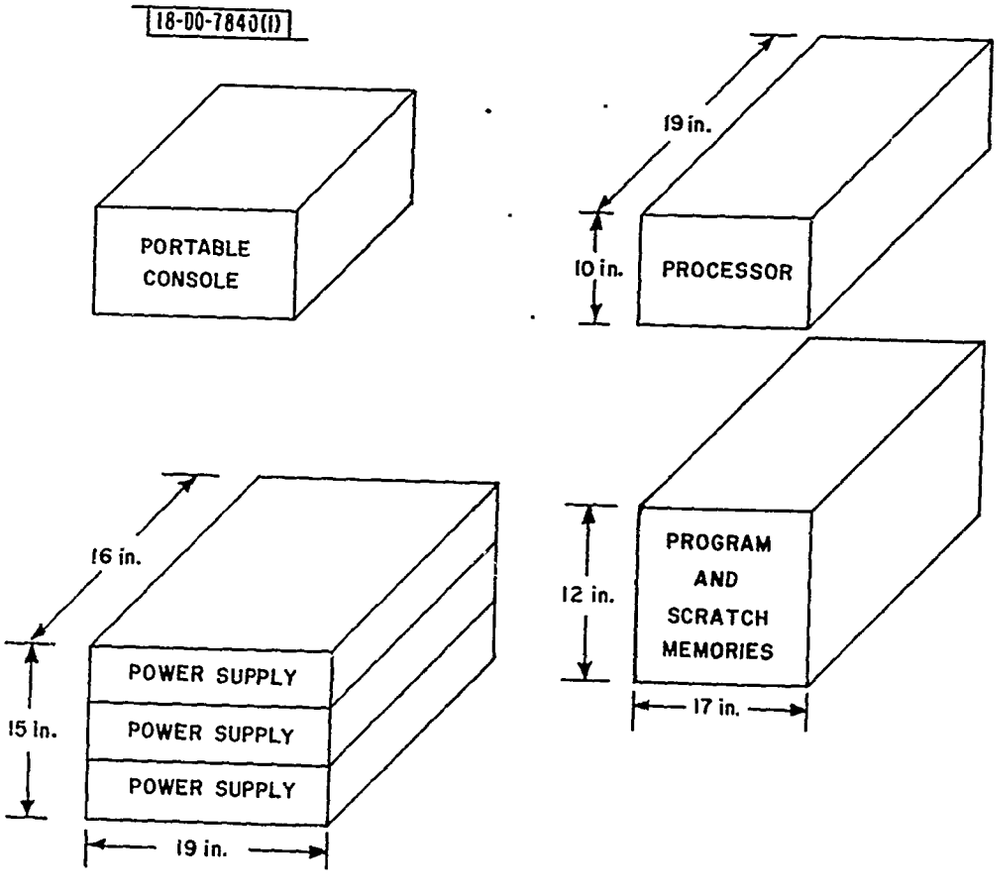


Fig. 35. Tentative system outline.

REFERENCES

1. B. Gold, I. Lebow, P. McHugh, C. Rader, "The Fast Digital Processor – a Programmable High Speed Signal Processing Computer," IEEE Trans. on Computers (January 1971).

2. G. D. Hornbuckle, E. I. Ancona, "The LX-1 Microprocessor and Its Application to Real-Time Signal Processing," IEEE Trans. on Computers (August 1970).

3. H. W. Gschwind, Design of Digital Computers, Springer-Verlag, Austria, 1967, pp. 235-243.

APPENDIX

ASP PROGRAMMING INSTRUCTIONS

I. Adds/Subtracts

Format I

6	6	6	6
Op Code	A	B	D

$\emptyset\emptyset, 1 \quad [A_\mu] \pm [B_\mu] \rightarrow [D_\mu]$ *†‡

$\emptyset 2, 3 \quad [A_\mu] \pm [B_\ell] \rightarrow [D_\mu]$

$\emptyset 4, 5 \quad [A_\ell] \pm [B_\ell] \rightarrow [D_\ell]$

$\emptyset 6, 7 \quad [A_\ell] \pm [B_\mu] \rightarrow [D_\ell]$

$1\emptyset, 11$

$12, 13 \quad [A_\mu] \pm [B_\mu] \rightarrow [D_\mu] \; ; \; [A_\ell] \pm [B_\ell] \rightarrow [D_\ell]$

$14, 15 \quad [A_\mu] \pm [B_\ell] \rightarrow [D_\mu] \; ; \; [A_\ell] \pm [B_\mu] \rightarrow [D_\ell]$

$16, 17 \quad \frac{1}{2}([A_\mu] \pm [B_\mu]) \rightarrow [D_\mu] \; ; \; \frac{1}{2}([A_\ell] \pm [B_\ell]) \rightarrow [D_\ell]$

$2\emptyset, 21 \quad [A] \pm [B] \rightarrow [D]$, DOUBLE PRECISION

Format II

6	6	6	6
Op Code	Sub Op Code	B	D

$77 \quad \emptyset\emptyset, 1 \quad \frac{1}{2}([D_\mu] \pm [B_\ell]) \rightarrow [D_\mu]; \; \frac{1}{2}([D_\ell] \pm [B_\mu]) \rightarrow [D_\ell]$

$77 \quad \emptyset 2, 3 \quad 2([D_\mu] \pm [B_\mu]) \rightarrow [D_\mu]; \; 2([D_\ell] \pm [B_\ell]) \rightarrow [D_\ell]$

$77 \quad \emptyset 4, 5 \quad 2([D_\mu] \pm [B_\ell]) \rightarrow [D_\mu]; \; 2([D_\ell] \pm [B_\mu]) \rightarrow [D_\ell]$

$77 \quad \emptyset 6, 7 \quad |[B_\mu]| \rightarrow [D_\mu]; \; |[B_\ell]| \rightarrow [D_\ell]$

$77 \quad 1\emptyset, 11 \quad [D] \pm [B_\ell] \rightarrow [D]$

$77 \quad 12, 13 \quad 2([D] \pm [B_\ell]) \rightarrow [D]$

$77 \quad 14, 15 \quad [D] \pm [B_\mu] \rightarrow [D]$

$77 \quad 16, 17 \quad \frac{1}{2}([D] \pm [B_\mu] \rightarrow [D]$

$77 \quad 2\emptyset, 21 \quad 2([D] \pm [B]) \rightarrow [D]$

$77 \quad 22, 23 \quad \frac{1}{2}([D] \pm [B]) \rightarrow [D]$

} DOUBLE PRECISION

---

* All arithmetic is 2's complement.

† Subscripts μ and ℓ refer to upper and lower bytes, respectively.

‡ [X] refers to "contents of X."

II. Logical Operations

Format I

6	6	6	6
Op Code	A	B	D

22 $[A] \wedge [B] \rightarrow [D]$
23 $[A] \vee [B] \rightarrow [D]$
24 $[A] \oplus [B] \rightarrow [D]$

} 24 bits

Format II

6	6	6	6
Op Code	Sub Op Code	B	D

77 24 $\overline{[B]} \rightarrow [D]$, 24 bit.
77 25 $\overline{[B_\ell]} \rightarrow [D_\mu]$; $\overline{[B_\mu]} \rightarrow [D_\ell]$
77 26 $\overline{[B_\mu]} \rightarrow [D_\mu]$; $\overline{[B_\ell]} \rightarrow [D_\ell]$

III. Multiplication Operations

6	6	6	6
Op Code	A	B	D

25 $[A_\mu] \times [B_\mu] \rightarrow [D_\mu]$
26 $[A_\mu] \times [B_\ell] \rightarrow [D_\mu]$
27 $[A_\ell] \times [B_\ell] \rightarrow [D_\ell]$
30 $[A_\ell] \times [B_\mu] \rightarrow [D_\ell]$
31 $[A_\mu] \times [B_\mu] \rightarrow [D_\mu]$; $[A_\ell] \times [B_\ell] \rightarrow [D_\ell]$
32 $[A_\mu] \times [B_\ell] \rightarrow [D_\mu]$; $[A_\ell] \times [B_\mu] \rightarrow [D_\ell]$

} Fraction multiplies: Bits 12 - 23 of product outputted.

33 $[A_\mu] \times [B_\mu] \rightarrow [D_\mu]$; $[A_\ell] \times [B_\ell] \rightarrow [D_\ell]$ } Integer multiply: Bits 1 - 12 of product outputted.

34 $[A_\ell] \times [B_\ell] \rightarrow [D]$
35 $[A_\ell] \times [B_\mu] \rightarrow [D]$

} Full multiplies: All product bits are outputted.

IV. Division Operations

6	6	6	6
Op Code	A	B	D

36 $[A] \div [B_\mu] \rightarrow [D_\mu]$
37 $[A] \div [B_\ell] \rightarrow [D_\mu]$

} Remainder on D_ℓ.

V. Square Root Function

6	6	6	6
Op Code	Sub Op Code	B	D

77 27 $([A])^{1/2} \rightarrow [D_\mu]$

VI. Constants

6	12	6
Op Code	Y	D

40 $Y \rightarrow [D_\mu]$

41 $Y \rightarrow [D_\ell]$

42 $Y + [D_\mu] \rightarrow [D_\mu]$

43 $Y + [D_\ell] \rightarrow [D_\ell]$

Y is 11 bits plus sign.

VII. Special Functions

6	6	6	6
Op Code	Sub Op Code	B	D

77 30 $[D_\ell] + BRV([B_\ell]) \rightarrow [D_\ell]$, Bit-Reversed Add. Bit reverse of $[B_\ell]$ added to $[D_\ell]$, carry propagates left to right.

77 31 $(N - 1) \rightarrow [D_\ell]$, N = number leading 1s or 0s in $[B_\mu]$. Scale Function, for normalization.

77 32 $2^{[B_\ell]} \rightarrow [D_\ell]$, Positive Scale Factor. Left shifting. (SFACP).

77 33 $2^{11-[B_\ell]} \rightarrow [D_\ell]$, Negative Scale Factor. Right shifting. (SFACN).

77 34 $\frac{1}{2}[B_\ell] \rightarrow [D_\ell]$, Zero shifted into bit 12. For double-precision multiplies (ZINJ).

VIII. Memory Reference Ops

6	6	6	6
Op Code	Sub Op Code	B	D

77 35 $[M_s(B_\mu)] \rightarrow [D_\mu]$

77 36 $[M_s(B_\ell)] \rightarrow [D_\mu]$

77 37 $[M_s(B_\mu)] \rightarrow [D_\ell]$

77 40 $[M_s(B_\ell)] \rightarrow [D_\ell]$

77 41 $[D_\mu] \rightarrow [M_s(B_\mu)]$

77 42 $[D_\mu] \rightarrow [M_s(B_\ell)]$

77 43 $[D_\ell] \rightarrow [M_s(B_\mu)]$

77 44 $[D_\ell] \rightarrow [M_s(B_\ell)]$

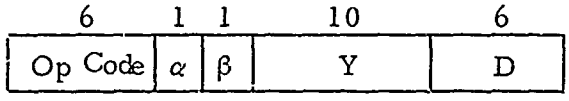
} 12-bit transfers.

VIII. Memory Reference Ops (continued)

77 45 $[M_s(B_\mu)] \rightarrow [D]$
77 46 $[M_s(B_\ell)] \rightarrow [D]$
77 47 $[D] \rightarrow [M_s(B_\mu)]$
77 5Ø $[D] \rightarrow [M_s(B_\ell)]$

} 24-bit transfers.

IX. Arithmetic Branches



Y: Jump destination

α: If set, skip next instruction if jump occurs (SOJ)

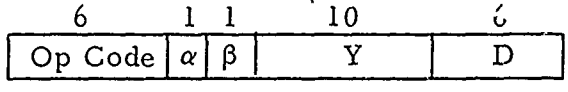
β: If set, test upper byte. Else test lower.

Return point: $P + 2 \rightarrow R1_\mu$

44 JPR: Jump if $[D_{\mu,\ell}] > Ø$
45 JNR: Jump if $[D_{\mu,\ell}] < Ø$
46 JZR: Jump if $[D_{\mu,\ell}] = Ø$
47 JUZR: Jump if $[D_{\mu,\ell}] \neq Ø$
5Ø JPZR: Jump if $[D_{\mu,\ell}] \geq Ø$
51 JNZR: Jump if $[D_{\mu,\ell}] \leq Ø$
52 JZRD: Jump if $[D] = 0$.
53 JUZRD: Jump if $[D] \neq 0$.

} Test both bytes always, β not used.

X. Unconditional Branches

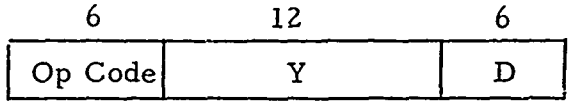


Y: Jump destination

α: If set, skip next instruction when jump occurs.

54 JPS: Jump to Y and save P + 2 in $[D_{\mu,\ell}]$ as specified by β.

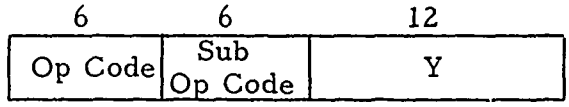
55 XJP: Jump to Y + $[D_{\mu,\ell}]$ as specified by β. Save P + 2 in $R1_\mu$



56 JP: Jump to Y and save P + 2 in $R1_\mu$.
Skip next instruction.

57 RJP: Jump to $[D_\ell]$ and write Y into $[D_\ell]$.
Skip next instruction. Used for closing interrupt service routines. Y is entry point.

XI. Overflow Branches



Notes:
1) There is an overflow flag for each D-bus byte.
2) All overflow jumps save P + 2 in $R1_\mu$.
3) Flags may be set by following single precision ops:
   a) Add or subtract
   b) Magnitude function
   c) Left shifts
   d) Multiplies
   e) Divisions (upper byte only)
   f) Square root, if operand negative. (Upper byte.)
4) Upper byte flag only can be set by double precision ops:
   a) Adds or subtracts
   b) Left shifts
5) Control transferred to Y.

77 51 CLOV: Clear all overflow flags.

77 52 JOVL: Jump on lower byte overflow and clear flag. Next instruction always executed.

77 53 JOVLS: Jump on lower byte overflow and clear flag. Next instruction skipped if jump occurs.

77 54 JOVU: Jump on upper byte overflow and clear flag. Next instruction always executed.

77 55 JOVUS: Jump on upper byte overflow and clear flag. Next instruction skipped if jump occurs.

77 56 JOVUL: Jump if either overflow set, do not clear flags. Next instruction always executed.

77 57 JOVULS: Jump if either overflow set and do not clear flags. Next instruction skipped if jump occurs.

XII. Input/Output

6	6	6	2	1	3
Op Code	A	B	γ	μ	σ

6Ø DMA: Initiate automatic input/output sequence according to the following rules:

a) $\underline{\sigma}$ selects 1 of 8 channels. Channels 6 and 7 are direct memory access data channels. Channels 0 - 5 are control channels and have no associated data paths.

b) $\underline{\gamma}$ selects I-O function desired:

0 - Input request

1 - Input status request

2 - Output request

3 - External function request

c) $\underline{\mu}$ is the monitor interrupt. Main program is interrupted when I-O buffer is complete.

d) $\underline{A}, \underline{B}$ select general registers which are interpreted as follows:

A:

12	12
Size of Data Block	Interrupt Service Return Entry Point

B:

12	12
Increment	M_s Starting Address

6	12	3	3
Op Code	Y	γ	σ

61 IOJP: Jump to Y if the condition specified by γ is met by the channel selected by σ. Save P + 2 in $R1_{\mu}$. Skip next instruction if jump occurs.

γ provides for the following tests:

- ∅ - Input inactive
- 1 - Input status request inactive
- 2 - Output inactive
- 3 - External function request inactive
- 4 - Input or output inactive
- 5 - Input status request or external function request inactive
- 6 - Input active
- 7 - Output active

XIII. Block Transfer

6	6	6	1	5
Op Code	A	B	α	

62 BLOK: Transfer a list of words between M_s and M_p. Machine is effectively stopped. Program execution resumes after BLOK is complete with the first instruction subsequent to the BLOK prior to M_p modification. A and B select general registers which are interpreted as follows:

A:

12	12
Size of Data Block	Starting M_p Address

B:

12	12
Increment	Starting M_s Address

α = ∅: M_s to M_p.
α = 1: M_p to M_s.

6	6	8	1	1	1	1
Op Code	Sub Op Code	Not Used	S_4	S_3	S_2	S_1

77 77 Stop on switches - STPS: Stop the computer if the combination of stop switch settings delineated by $S_{1,2,3,4}$ is encountered. If all S bits are set, computer halts unconditionally. Normally, only one S bit is set.

Note: 1) 52 of the 64 6-bit Op Codes have been assigned.

2) 53 of the 64 12-bit Op Codes have been assigned.

3) These are only tentative assignments.